THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

AUTOMATIC EMPIRICAL TECHNIQUES FOR DEVELOPING EFFICIENT MPI COLLECTIVE COMMUNICATION ROUTINES

By

AHMAD FARAJ

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Summer Semester, 2006

The members of the Committee approve the Dissertation of Ahmad Faraj defended on July 7, 2006.

Xin Yuan
Professor Directing Dissertation

Guosheng Liu
Outside Committee Member

David Whalley
Committee Member

Kyle Gallivan
Committee Member

Ashok Srinivasan
Committee Member

Approved:

David Whalley, Chair
Department of Computer Science

Professor Supremo, Dean, College of Arts and Sciences

The Office of Graduate Studies has verified and approved the above named committee members.

This dissertation is dedicated to the very special person in my life: my lovely wife, Mona. She shared all moments I went through during the journey of pursuing my doctorate, giving me comfort whenever I was frustrated and being proud of me whenever I made an accomplishment. I am very grateful for her patience, her support, her companionship, and above all, her endless love…

ACKNOWLEDGEMENTS

I would like to recognize, acknowledge, and thank Professor Xin Yuan for giving me the opportunity to share with him such an amazing journey in the research area we have worked in together as well as for the advisement and help in writing this dissertation. It was a great honor working with him. I must also acknowledge and thank the members of my dissertation committee: David Whalley, Kyle Gallivan, Ashok Srinivasan, and Guosheng Liu. I am grateful to many friends and colleagues who assisted, advised, and supported my research and writing efforts over the years. Especially, I need to express my gratitude and deep appreciation to Stephen Hines and Stephen Carter. Special thanks to my family, the Carter family, and all who dedicated and supported me through this stage of my life.

TABLE OF CONTENTS

List of Tables vii

List of Figures ix

Abstract xii

1. INTRODUCTION 1
- 1.1 Topology-specific Communication Algorithms for Ethernet Switched Clusters 3
- 1.2 Delayed Finalization of MPI Collective Communication Routines 4
- 1.3 Organization of Dissertation 6

2. RELATED WORK 7
- 2.1 Characteristics of MPI Applications 7
- 2.2 Optimizations of the MPI Library 8
- 2.3 Empirical Optimization Techniques 10

3. TOPOLOGY-SPECIFIC ALGORITHMS FOR ETHERNET SWITCHED CLUSTERS 12
- 3.1 Network Model & Terminology 12
- 3.2 All-to-All Personalized Communication 14
- 3.3 All-to-All Broadcast Operation 38
- 3.4 Broadcast Operation 61
- 3.5 Summary 78

4. DELAYED FINALIZATION OF MPI COLLECTIVE COMMUNICATION ROUTINES 80
- 4.1 STAGE-MPI 82
- 4.2 Application Behavior and Its Impacts on Collective Operations 107
- 4.3 STAR-MPI 129
- 4.4 Summary 150

5. CONCLUSION & FUTURE WORK 152
- 5.1 Conclusion 152
- 5.2 Future Work 152

REFERENCES 155

BIOGRAPHICAL SKETCH 162

LIST OF TABLES

3.1 Phases from ring scheduling 18

3.2 Rotate pattern for realizing $t_i \rightarrow t_j$ when $|M_i| = 6$ and $|M_j| = 4$ 21

3.3 Mapping between senders and the receivers in Step 2 22

3.4 Results of global and local message assignment for the cluster in Figure 3.2 . 26

3.5 Message scheduling based AAPC schemes used in the evaluation 30

3.6 LR1 vs. LR2 on topology (4) 59

3.7 The performance of broadcast algorithms for very large messages 65

3.8 Completion time of different all–to–all algorithms 79

4.1 Tuned MPI_Alltoall, MPI_Allgather, and MPI_Allreduce 99

4.2 LAM/MPI and MPICH algorithms for MPI_Alltoall, MPI_Allgather, and MPI_Allreduce 100

4.3 Tuning time 100

4.4 Execution time (seconds) 105

4.5 Performance of applications on topology (a) using collective routines tuned with different timing mechanisms 106

4.6 One way point-to-point communication time on Lemieux 111

4.7 One way point-to-point communication time on the Beowulf cluster 111

4.8 Summary of benchmarks (e. time is execution time while c. time is communication time) 112

4.9 The dynamic counts of major collective communication routines in the benchmarks ($n = 64$) 113

4.10 The average of the worst/average case imbalance factors among all collective routines on Lemieux 114

4.11 The average of the worst/average case imbalance factors among all collective routines on the Beowulf cluster 115

4.12 The imbalance factor for major collective routines in the benchmarks 118

4.13 Basic AEOS vs. enhanced AEOS on topology (a) 142

4.14 Collective routines in the applications on different topologies 143

4.15 Communication algorithms used in STAR, STAGE, and MPICH on different topologies (T) 144

4.16 Application completion times on different topologies 145

4.17 Application completion times 146

4.18 Per invocation time (ms) for collective operations in the micro-benchmark . 147

4.19 Per invocation overheads (in millisecond) for collective operations in the micro-benchmark 148

LIST OF FIGURES

3.1 An example Ethernet switched cluster 13

3.2 Identifying the root in an example cluster 17

3.3 A two level view of the network 17

3.4 Global message scheduling for the example in Figure 3.2 19

3.5 The global and local message assignment algorithm 23

3.6 Code segment for measuring the performance of MPI_Alltoall 32

3.7 Topologies used in the evaluation 32

3.8 The performance of different AAPC implementations 34

3.9 Sender-based synchronization versus receiver-based synchronization 35

3.10 Message scheduling based schemes with different synchronization mechanisms 35

3.11 Partially synchronized schemes with sender-based pair-wise synchronizations 36

3.12 Schemes with barrier synchronizations 36

3.13 Performance of *Sender all* and *tuned scheduling based* 37

3.14 Performance of different fully synchronized schemes for large messages 38

3.15 All-to-all Broadcast on 4 nodes 38

3.16 DFS numbering 43

3.17 Three cases 45

3.18 Constructing the logical array for an intermediate switch 48

3.19 Logical rings from Algorithm 1 and Algorithm 2 49

3.20 The relationship between the optimal logical ring and logical arrays for subtrees 52

3.21 Code segment for measuring MPI_Allgather performance 55
3.22 Topologies used in the experiments 56
3.23 Performance of LR1, LAM, and MPICH-LAM on topologies (1) to (4) 57
3.24 Performance of LR1, LAM, and MPICH-LAM on topologies (5) and (6) . . . 58
3.25 Performance of LR1 and LR3 with different tree construction methods . . . 61
3.26 Examples of broadcast trees 64
3.27 DFS numbering 67
3.28 Heuristic to compute contention-free binary trees (Algorithm 5) 70
3.29 Performance of Algorithm 5 72
3.30 Topologies used in the evaluation 73
3.31 Code segment for measuring MPI_Bcast performance 73
3.32 Performance of pipelined broadcast with different broadcast trees on topology (1) 74
3.33 Performance of different broadcast algorithms (medium sized messages) . . . 75
3.34 Performance of different broadcast algorithms (large sized messages) 76
3.35 Contention-free broadcast trees versus topology unaware broadcast trees . . 77
3.36 Pipelined broadcast using linear trees with different segment sizes 77
3.37 Example topology 78
4.1 High level view of a DF library 81
4.2 System overview 83
4.3 Three different built-in timing mechanisms in STAGE-MPI 94
4.4 A tuning process example 96
4.5 Examples of tuned routines 97
4.6 Topologies used in the experiments 98
4.7 MPI_Alltoall on topology (a) 102
4.8 MPI_Alltoall on topology (b) 103
4.9 MPI_Allgather on topology (c) 103

4.10 MPI_Allreduce on topology (c) 104

4.11 Process arrival pattern 110

4.12 The distribution of worst case imbalance factors on the two platforms 116

4.13 The distribution of worst case imbalance factors in each benchmark for operations with small message sizes ($\leq$ 1000 Bytes) on both platforms 117

4.14 The distribution of worst case imbalance factors in each benchmark for operations with medium and large message sizes ($>$ 1000 Bytes) on both platforms 118

4.15 The imbalance factors for major collective routines in FT and NBODY on Lemieux with 128 processors 119

4.16 Code segment for a micro-benchmark 120

4.17 Process arrival patterns in the micro-benchmark (64KB message size, 200ms computation time) on the two platforms 121

4.18 Computation imbalance versus arrival pattern imbalance on the two platforms 122

4.19 Computation imbalance versus total computation time 123

4.20 Code segment for measuring the impacts of imbalanced process arrival patterns124

4.21 MPI_Alltoall on Lemieux (32 processors) 126

4.22 MPI_Alltoall on the Beowulf cluster 126

4.23 High level view of STAR-MPI 130

4.24 Using STAR-MPI algorithm to tune MPI_Alltoall 136

4.25 Topologies used in the experiments 140

4.26 An example micro-benchmark 141

4.27 All-to-all micro-benchmark results on 128 processors (Lemieux), average per invocation time 149

ABSTRACT

Due to the wide use of collective operations in Message Passing Interface (MPI) applications, developing efficient collective communication routines is essential. Despite numerous research efforts for optimizing MPI collective operations, it is still not clear how to obtain MPI collective routines that can achieve high performance *across platforms and applications*. In particular, while it may not be extremely difficult to develop an efficient communication algorithm for a given platform and a given application, including such an algorithm in an MPI library poses a significant challenge: the communication library is general-purpose and must provide efficient routines for different platforms and applications.

In this research, a new library implementation paradigm called *delayed finalization of MPI collective communication routines* (DF) is proposed for realizing efficient MPI collective routines across platforms and applications. The idea is to postpone the decision of which algorithm to be used for a collective operation until the platform and/or application are known. Using the DF approach, the MPI library can maintain, for each communication operation, an extensive set of algorithms, and use an automatic algorithm selection mechanism to decide the appropriate algorithm for a given platform and a given application. Hence, a DF based library can adapt to platforms and applications.

To verify that the DF approach is effective and practical, Ethernet switched clusters are selected as the experimental platform and two DF based MPI libraries, STAGE-MPI and STAR-MPI, are developed and evaluated. In the development of the DF based libraries, topology-specific algorithms for all-to-all, all-gather, and broadcast operations are designed for Ethernet switched clusters. The experimental results indicate that both STAGE-MPI and STAR-MPI significantly out-perform traditional MPI libraries including LAM/MPI and MPICH in many cases, which demonstrates that the performance of MPI collective library routines can be significantly improved by (1) incorporating platform/application specific communication algorithms in the MPI library, and (2) making the library adaptable to platforms and applications.

CHAPTER 1

INTRODUCTION

A key building block of exploiting clusters of workstations, which have recently become popular as high performance computing (HPC) platforms, is the Message Passing Interface (MPI) that enables communication within the clusters. The standardization of MPI [51] has facilitated the development of applications that use message passing as the programming paradigm and has resulted in a large number of MPI applications. In order to deliver high performance to these MPI applications, it is imperative that the library implementation of MPI realizes the communications efficiently.

An important functionality provided by the MPI standard is the support for collective communications. In a collective communication operation, multiple processors (or nodes) participate in the operation, as opposed to only two processors (sender and receiver) in a point–to–point communication operation. Studies have indicated that MPI collective operations are used in most MPI applications and account for a significant portion of the total time in some applications [65]. Despite the importance of MPI collective communication operations and the research efforts that sought to optimize them, it is still not clear how to realize MPI collective communication routines that can deliver high performance **across platforms and applications**.

Traditionally, MPI library developers must decide at the library design time which communication algorithm is to be used for a given collective operation in a given situation, such as a given message size or a given number of processors. Such a library implementation paradigm has many limitations. First, for the libraries to work in all cases, the algorithms included in these libraries have to be general-purpose since much of the platform information is unavailable at the library design time. For instance, the topology of the platform is usually unknown and thus topology-specific communication algorithms cannot be included in the library. However, such platform-specific (e.g. topology-specific) algorithms can usually achieve much higher performance than general-purpose algorithms [15, 16, 17, 18, 24, 25]. Second, there are many important platform parameters that can significantly affect the performance of a communication algorithm. These include OS context switching overheads, nodal architecture, ratio between the network and the processor speeds, switch design, amount of buffer memory in switches, and network topology. Since these parameters differ from one platform to another, it is impossible for the library developer to make the right choices on the communication algorithms across different platforms. As a result, the selection of communication algorithms in traditional libraries is somewhat inadequate. Third, the application behavior is not considered in the development of library routines as the collective algorithms are decided before the application is known. MPI collective routines can be used in different program contexts, which may require different algorithms to achieve the best performance (for different contexts).

MPI developers have recognized these issues and have tried to remedy the problems by developing adaptive MPI collective routines that allow different algorithms to be used for an operation based on the message size and/or number of processors. However, due to the limitation of the library implementation paradigm, where communication algorithms are decided at the library design time, the software adaptability of these libraries, including LAM/MPI [44] and MPICH [52], is limited. The routines may adapt only to message sizes and the number of processors, but cannot adapt to platforms and applications. Hence, although the problems described above are somewhat alleviated, they remain unsolved in these libraries.

There are two essential issues for implementing efficient MPI collective communication routines across platforms and applications. The first issue is to develop and include platform/application specific algorithms in the library. This creates a fundamental dilemma: on one hand, platform/application specific algorithms are designed for specific situations; on the other hand, a communication library is usually general-purpose and must work in different situations. Even if such algorithms were included in the library, the second issue that is of a great importance is the ability of the MPI library to adapt to platforms and applications by using the most efficient algorithms, including platform/application specific and platform/application unaware, to carry out an operation in a given situation (platform and application). Clearly, traditional MPI libraries such as LAM/MPI [44] and MPICH

[52] fail to address either of the two issues: the algorithms included in these libraries are all platform/application unaware, and the library adaptability is limited. As a result, these libraries are unable to deliver high performance for applications on many platforms [9].

The thesis of this dissertation is that (1) it is possible to develop an adaptive MPI library whose collective routines can adapt to platforms and applications, and more importantly, (2) the performance of MPI collective library routines can be significantly improved by incorporating platform/application specific algorithms and making the library adaptive to the platform/application. To verify this thesis, Ethernet switched clusters are selected as the experimental platform and efficient adaptive MPI libraries are developed for this platform. The reasons for choosing Ethernet clusters are that Ethernet is the most widely used networking technology and that accesses to Ethernet switched clusters were available at the time this research was conducted.

The contributions of this research can be classified into two areas. First, platform-specific communication algorithms are developed for Ethernet switched clusters [15, 16, 17, 18, 24, 25]. Topology-specific communication algorithms for commonly used collective communication routines, including *MPI_Alltoall*, *MPI_Allgather*, and *MPI_Bcast*, are designed, implemented, and evaluated. Second, a new paradigm for implementing MPI collective routines, called the *delayed finalization of MPI collective communication routines* (DF) approach, is proposed. The DF approach allows for more software adaptability than that of traditional libraries. In particular, it allows platform-specific algorithms to be included in the library, and it is able to find efficient algorithms for a given platform and/or application. To study the effectiveness of the DF approach and investigate the difficulties in developing a DF library, two DF library prototypes are designed, implemented, and evaluated [19, 20, 21, 22]. In the following, these contributions are discussed in details.

1.1 Topology-specific Communication Algorithms for Ethernet Switched Clusters

The first step towards obtaining high performance collective communication routines is to design algorithms that are optimized for the particular platform on which they will run so that communications can be carried out efficiently. Such algorithms are usually architecture-dependent. A major issue in Ethernet switched clusters is network contention. The physical topology of an Ethernet switched cluster is a tree [74], which has a limited bisection width.

Without considering the network topology, it is very likely that two or more messages in a collective operation use the same link at the same time, which causes contention and degrades the performance. In general, using topology-unaware collective communication algorithms can perform reasonably well when the message size is small since the network can handle such cases without significantly degrading the performance. However, for collective communications with large messages, the network contention problem can significantly affect the communication performance. This is particularly true when the number of nodes is large enough that the nodes must be connected to multiple switches. In this case, it is essential to develop topology-specific collective communication algorithms that consider the network topology to eliminate network contention.

In this work, topology-specific communication algorithms for efficiently realizing the all–to–all, all–gather, and broadcast operations on Ethernet switched clusters are developed [15, 16, 17, 18, 24, 25]. To evaluate these algorithms, automatic routine generators are implemented. These generators take the topology information as input and generate topology-specific MPI collective communication routines. The performance evaluation of the generated routines for the different collective operations shows that in many cases the topology-specific algorithms achieve significantly higher performance than the topology-unaware algorithms included in LAM/MPI [44] and MPICH [52] libraries.

Although the topology-specific communication algorithms are proposed for Ethernet switched clusters with physical tree topologies, many of the proposed algorithms can be used in other networks since a tree topology can be embedded in most connected networks. This will require finding the tree embedding with a spanning tree algorithm and then applying the proposed topology-specific communication algorithms (for the tree topology).

1.2 Delayed Finalization of MPI Collective Communication Routines

The idea of the *delayed finalization of MPI collective communication routines* (DF) approach is to postpone the decision of which algorithm to be used for a collective operation until the platform and/or the application are known. This potentially allows architecture and/or application specific optimizations to be applied. There are two major components in a DF library: (1) an algorithm repository and (2) an automatic algorithm selection mechanism. The algorithm repository contains, for each operation, an extensive set of platform-unaware and platform-specific algorithms that can potentially achieve high performance in different situations. The automatic selection mechanism evaluates the communication performance of different algorithms (from the repository) and selects the best communication algorithms to be used in the final routine. Since there are many factors, including platform parameters and application behavior, that can affect the performance of communication algorithms, and since some factors are hard to model or measure in practice, developing an effective automatic algorithm selection mechanism can be challenging. This thesis considers using an empirical technique in the algorithm selection (also called **tuning**) process: different communication algorithms are executed, the performance of each algorithm is measured, and the best algorithm is chosen based on the performance measurements. The empirical approach allows the aggregate effects of all factors that affect the performance to be used in the algorithm selection process since the aggregate effects are summarized in the measurement results.

The implementation paradigm for a DF based library is different from that for the traditional MPI libraries: the DF library developers only implement the communication algorithms and the mechanisms to select the algorithms, but do not make decisions about which algorithms to use for an operation. The final communication algorithms for an operation are automatically selected by the algorithm selection mechanism, which may take platform architecture and application into account.

To study the effectiveness of the DF approach and investigate the difficulties in developing a DF library, two prototype DF libraries are designed, implemented, and evaluated. The first DF prototype library is the **S**tatic **T**uning and **A**utomatic **G**eneration of **E**fficient **MPI** collective routines (**STAGE-MPI**) system [20], and the second DF prototype library is composed of **S**elf **T**uned **A**daptive **R**outines for **MPI** collective operations (**STAR-MPI**) [22]. Both STAGE-MPI and STAR-MPI maintain an extensive algorithm repository but differ in the tuning process. In STAGE-MPI, the tuning process occurs at the library installation time. For each collective operation, the performance of different algorithms is measured across different message sizes using a standard performance measurement scheme such as Mpptest [32], and the best performing algorithm is decided for each message size. The system then determines a message size range for each of the best performing algorithms and automatically generates the final tuned routine, which may include different communication algorithms for different message size ranges. In STAR-MPI, the tuning process occurs dynamically at run-time in the context of application execution. Each time a STAR-MPI routine is invoked in an application, an algorithm from the repository is used to realize the invocation and the performance of the algorithm is measured. After a sufficient number of invocations of the routine, the best performing algorithm is decided and used in subsequent invocations.

In STAGE-MPI, the tuning process can be very lengthy. Thus, the system is applicable only when the platform is fixed and the routines are repeatedly invoked. Moreover, since STAGE-MPI runs on the platform where the MPI library is to be installed and used, the system selects efficient communication algorithms that are tuned to the given platform; that is, the final routines practically adapt to the platform but not the applications. The STAR-MPI library is designed for MPI programs where collective communication routines are called iteratively a large number of times. STAR-MPI is applicable and well-suited for typical supercomputing clusters where users are given a different partition every time they run a job. Furthermore, as STAR-MPI routines run in the context of the application on the platform, the routines are able to select communication algorithms that perform best for the application on the platform. Thus, STAR-MPI routines can adapt to both the application and platform.

The performance studies of STAGE-MPI and STAR-MPI indicate that both out-perform traditional MPI libraries, including LAM/MPI [44] and MPICH [52], in many cases, which demonstrates the effectiveness of the DF approach. By incorporating platform/application specific communication algorithms in the library and making the library adaptive to the platform/application with the DF approach, the performance of MPI collective routines can be significantly improved over traditional MPI libraries.

1.3 Organization of Dissertation

The rest of this dissertation is organized as follows. Chapter 2 presents the related work. Chapter 3 describes in details the topology-specific communication algorithms for the three MPI collective routines (*MPI_Alltoall*, *MPI_Allgather*, and *MPI_Bcast*). Chapter 4 discusses the DF approach and libraries. Finally, Chapter 5 presents the conclusion and future work.

CHAPTER 2

RELATED WORK

The research mostly related to this work falls into three broad areas: analysis of the characteristics of MPI applications, optimizations of the MPI library, and empirical optimization techniques. These related areas are described next.

2.1 Characteristics of MPI Applications

Over the years, parallel applications and parallel architectures have evolved. Since MPI became a standard [50], a large number of MPI based parallel applications have been developed. Analyzing the characteristics of MPI applications is essential to develop efficient parallel systems. Some of the early work [1, 8] have focused on the use of MPI applications for evaluation purposes. Work in [1] evaluates the overall performance of a large number of parallel architectures using the NAS application benchmarks [54]. The study in [8] uses the NAS benchmarks to evaluate two communication libraries over the IBM SP machine.

Recently, the MPI community has recognized the need to have efficient implementations of the MPI standard. Characterizing and understanding the application behavior is critical for developing an efficient MPI library, which is evident in numerous research efforts focusing on analyzing MPI communication behavior [11, 23, 43, 71, 72, 80]. In [80], the performance of parallel applications is analyzed using a technique that automatically classifies inefficiencies in point-to-point communications. The study analyzes the usage of MPI collective communication routines and their elapsed times. The studies in [23, 43] performed quantitative measures of the static and dynamic MPI routines in scientific parallel applications. Work in [71] performed statistical analysis of all-to-all elapsed communication time on the IBM SP2 machine to understand the causes of performance drop as the number of processors increases. The researchers in [72, 11] examined the NAS parallel benchmarks

[54] to quantitatively describe the usage of MPI routines and the distribution of message sizes.

The analysis performed on the parallel applications in these studies often involves the investigation of communication attributes such as the type of MPI routines, message size, message volume, message interval, bandwidth requirement, and communication elapsed time. This thesis includes a study that focuses on a communication attribute for collective operations: the process arrival pattern. Since a collective operation involves multiple processes that can arrive at the routine call site at different times, the process arrival pattern defines the timing when different processes arrive at the collective operation. This communication attribute has a significant impact on the performance of collective communication routines.

2.2 Optimizations of the MPI Library

The success of the MPI standard can be attributed to the wide availability of two MPI implementations: LAM/MPI [44] and MPICH[52, 77]. Many researchers have worked on optimizing the MPI library [47, 38, 26, 40, 55, 69, 75, 77]. In [40], optimizations of MPI over Wide Area Networks by considering the network details are proposed. In [55], a compiler based optimization approach is developed to reduce the software overheads in the library. In [47], MPI point–to–point communication routines are optimized using a more efficient primitive (Fast Message). Optimizations for a thread based MPI implementation are proposed in [75]. Optimizations for clusters of SMPs are presented in [69]. A combined compiler and library approach was proposed in [38]. In [26], multicast group management schemes are proposed for MPI collective routines that are realized using multicast primitives.

Due to the importance of collective communications, many efficient collective communication algorithms have also been developed. This thesis considers three collective operations: *MPI_Alltoall*, *MPI_Allgather*, and *MPI_Bcast*. Existing algorithms for these operations are surveyed next.

2.2.1 Algorithms for MPI_Alltoall

The all-to-all operation is used in many high performance applications, including matrix transpose, multi-dimensional convolution, and data redistribution. A large number of optimal message scheduling algorithms for different network topologies with different network models were developed. Many of the algorithms were designed for specific network topologies that are used in the parallel machines, including hypercube [36, 79], mesh [6, 68, 70, 76], torus [34, 46], k-ary n-cube [79], and fat tree [14, 62]. Heuristic algorithms were developed for the all-to-all operation on irregular topologies [48]. A framework that realizes the all-to-all operation using indirect communications was reported in [37]. Efficient scheduling schemes for clusters connected by a single switch were proposed in [73]. Some of the algorithms in [73] were incorporated in the recent improvement of MPICH library [77]. The all–to–all operation on Ethernet switched clusters with one or more switches is a special communication pattern on a tree topology. Message scheduling for such cases has not been considered before.

2.2.2 Algorithms for MPI_Allgather

Another commonly used operation in MPI applications is the all-gather operation, also known as all–to–all broadcast. Similar to the all-to-all operation, many all-gather algorithms were designed for specific network topologies that are used in the parallel machines, including hypercube [36, 79], mesh [84, 68, 76], torus [84], k-ary n-cube [79], fat tree [42], and star [56]. Work in [40] proposes optimizations of MPI collective operations, including all-gather, on wide area networks. The study in [30] investigates an efficient all-gather scheme on SMP clusters. Work in [85] explores the design of NIC-based all-gather with different algorithms over Myrinet/GM. Some efforts [73, 77] have focused on developing algorithms for different message sizes, and some of these algorithms have been also incorporated in the recent MPICH library [52]. In [7], the authors developed an efficient all-gather algorithm for small messages. The study in [4] compares a dissemination based all-gather with the recursive doubling algorithm [77] on Ethernet and Myrinet. In [35], nearly optimal all-gather schemes were developed for clusters connected by one or two switches. However, as indicated in [35] and [33], the algorithm proposed for the general topology in [35] is not always optimal for clusters connected by more than two switches.

2.2.3 Algorithms for MPI_Bcast

The broadcast operation in different environments has been extensively studied and a very large number of broadcast algorithms have been proposed. Algorithms developed for topologies used in parallel computers such as meshes and hypercubes (e.g. [36, 41]) are specific to the targeted topologies and platforms and cannot be applied to Ethernet switched clusters with physical tree topologies. Many researchers proposed the use of logical binomial trees for the broadcast operation and developed algorithms for computing contention-free binomial trees under different constraints [28, 39, 49]. In this thesis, pipelined broadcast tree algorithms are proposed to efficiently realize the broadcast operation on Ethernet switched clusters. Although atomic broadcast algorithms over physical tree topologies have also been developed [10, 63], such algorithms are different from pipelined broadcast algorithms.

Pipelined broadcast has been studied in different environments [2, 3, 61, 67, 78, 81]. In [81], an algorithm was designed to compute contention-free pipelined trees on the mesh topology. In [2, 3], heuristics for pipelined communication on heterogeneous clusters were devised. These heuristics focus on the heterogeneity of the links and nodes, but not the network contention, which is a major issue to consider in order to obtain efficient pipelined broadcast algorithms. The effectiveness of pipelined broadcast in cluster environments was demonstrated in [61, 67, 78]. It was shown that pipelined broadcast using topology-unaware trees can be very efficient for clusters connected by a single switch. The research in this thesis extends the work in [61, 67, 78] by considering clusters connected by multiple switches. In particular, methods for building contention-free trees for pipelined broadcast over a physical tree topology of multiple switches are developed and studied.

2.3 Empirical Optimization Techniques

Although researchers have developed many efficient communication algorithms to realize the different MPI collective operations, determining when to use one algorithm over another in a particular situation (e.g. message size, number of processors, platform, or application) requires an efficient selection mechanism. In both STAGE-MPI and STAR-MPI, the selection mechanism is a variation of the Automated Empirical Optimization of Software (AEOS) technique [82]. The idea of AEOS is to optimize software automatically using an empirical approach that includes timers, search heuristics, and various methods of software adaptability. This technique has been successfully applied to realize various computational library routines [5, 29, 82].

The technique in the STAGE-MPI system is closely related to the one used in the tuning system proposed in [78]. Both systems use a static AEOS technique to optimize collective communications. However, there are some significant differences. First, STAGE-

MPI considers algorithms that are specific to the physical topology while algorithms in [78] use logical topologies and are unaware of the physical topology. Second, the system in [78] tries to tune and produce common routines for systems with different numbers of nodes. STAGE-MPI is less ambitious in that routines for a specific physical topology are tuned. By focusing on a specific physical topology, STAGE-MPI is able to construct highly efficient routines. Third, [78] mainly focused on one-to-all and one-to-many communications and studied various message pipelining methods to achieve the best performance. In addition to one-to-all and one-to-many communications, STAGE-MPI also considers all–to–all and many–to–many communications where pipelining is not a major factor that affects the communication performance.

CHAPTER 3

TOPOLOGY-SPECIFIC ALGORITHMS FOR ETHERNET SWITCHED CLUSTERS

In Ethernet switched clusters, when the number of machines is large enough that they cannot connect to a single crossbar switch, multiple switches must be used to connect the machines. This reduces the network bisection width. In such a system, to eliminate network contention and achieve high performance, the network topology must be considered in the development of efficient communication algorithms. In this chapter, topology-specific communication algorithms for all–to–all, all–gather, and broadcast operations are introduced.

The chapter is organized as follows. Section 3.1 discusses the network model and terminology. Section 3.2 presents the topology-specific algorithm for the all–to–all personalized communication (*MPI_Alltoall*). Section 3.3 describes topology-specific algorithms for the all–to–all broadcast operation (*MPI_Allgather*). Section 3.4 presents the topology-specific algorithms for the broadcast operation (*MPI_Bcast*). Finally, Section 3.5 summarizes the chapter.

3.1 Network Model & Terminology

In Ethernet switched clusters, it is assumed that each workstation is equipped with one Ethernet port, and that each link operates in the duplex mode that supports simultaneous communications on both directions with the full bandwidth. Communications in such a system follow the 1-port model [3], 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 [74]. As a result, the physical topology of the network is always a **tree**.

The network can be modeled as a directed graph $G = (V, E)$ with nodes V corresponding to switches and machines, and edges E corresponding to unidirectional channels. Let S be the set of all switches in the network and M be the set of all machines in the network ($V = S \cup M$). Let $u, v \in V$, a directed **edge** $(u, v) \in E$ if and only if there is a link between node u and node v. The notion **link** (u, v) denotes the physical connection between nodes u and v. Thus, link (u, v) corresponds to two directed edges (u, v) and (v, u) in the graph. Since the network topology is a tree, the graph is also a tree: there is a unique path between any two nodes. For the tree topology, the switches are assumed to only be intermediate nodes while the machines can only be leaves. A switch as a leaf in the tree will not participate in any communication and, thus, can be removed from the graph. Also, it is assumed that there is at least one switch in the tree. Figure 3.1 shows an example cluster.

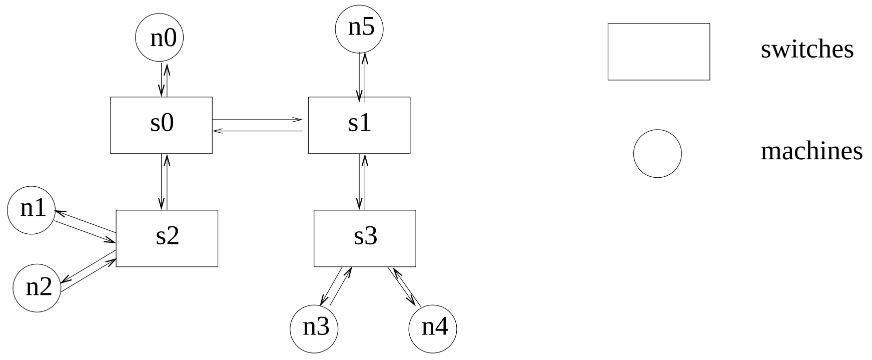


Figure 3.1: An example Ethernet switched cluster

The terminology used in this chapter is defined next. A *message*, $u \rightarrow v$, is data transmitted from node u to node v. A message is also called a *communication*. The notion *path*(u, v) denotes the set of directed edges in the unique path from node u to node v. For example, in Figure 3.1, *path*$(n0, n3) = \{(n0, s0), (s0, s1), (s1, s3), (s3, n3)\}$. The *path length* is defined as the number of switches a message travels through. For example, the path length of $n0 \rightarrow n3$ is 3. Two messages, $u_1 \rightarrow v_1$ 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, v_1) and $(x, y) \in$ *path*(u_2, v_2). A *pattern* is a set of messages. The notion $u \rightarrow v \rightarrow w \rightarrow ... \rightarrow x \rightarrow y$ is used to represent the pattern that consists of messages $u \rightarrow v$, $v \rightarrow w$, ..., and $x \rightarrow y$. A pattern is *contention free* if there is no contention between each pair of the communications in the pattern. A *phase* is a contention free pattern. The *load* on an edge is the number of times the edge is used in the pattern. The most loaded edge is called a *bottleneck* edge. For the all–to–all operation, the terms "the load of an edge (u, v)" and "the load of a link (u, v)" are the same. When discussing the all–to–all algorithm, the two terms will be used interchangeably. $|S|$ denotes the size of set S, B denotes the network bandwidth, and *msize* denotes the message size.

3.2 All-to-All Personalized Communication

All–to–all personalized communication (AAPC) is one of the most common communication patterns in high performance computing. In AAPC, each node in a system sends a different message of the same size to every other node. The Message Passing Interface routine that realizes AAPC is *MPI_Alltoall* [51]. AAPC appears in many high performance applications, including matrix transpose, multi-dimensional convolution, and data redistribution. Since AAPC is often used to rearrange the whole global array in an application, the message size in AAPC is usually large. Thus, it is crucial to have an AAPC implementation that can fully exploit the network bandwidth in the system.

This section presents a message scheduling scheme [15, 16] that theoretically achieves the maximum throughput of AAPC on any given Ethernet switched cluster. Similar to other AAPC scheduling schemes [34], the proposed scheme partitions AAPC into contention free phases. It achieves the maximum throughput by fully utilizing the bandwidth in the bottleneck links in all phases. Based on the scheduling scheme, an automatic routine generator is developed. The generator takes the topology information as input and produces an *MPI_Alltoall* routine that is customized for the specific topology. The automatically generated routine is compared with the original routine in LAM/MPI [44] and a recently improved *MPI_Alltoall* implementation in MPICH [77]. The results show that the automatically generated routine consistently out-performs the existing algorithms when the message size is sufficiently large, which demonstrates the superiority of the proposed AAPC algorithm in exploiting network bandwidth. In the following, the maximum aggregate throughput for AAPC is described, the proposed scheduling scheme is detailed, message scheduling based AAPC implementation issues are discussed, and the results of the performance evaluation study are reported.

3.2.1 Maximum Aggregate Throughput for AAPC

Since scheduling for AAPC when $|M| \leq 2$ is trivial, let us assume that $|M| \geq 3$. Note that the *load of AAPC pattern* is equal to the load of a bottleneck edge. Let edge (u, v) be one of the bottleneck edges for the AAPC pattern. Assume that removing link (u, v) partitions tree $G = (S \cup M, E)$ into two subtrees, $G_u = (S_u \cup M_u, E_u)$ and $G_v = (S_v \cup M_v, E_v)$. G_u is the connected component including node u, and G_v is the connected component including node v. AAPC requires $|M_u| \times |M_v| \times msize$ bytes data to be transferred across the link (u, v) in both directions. The best case time to complete AAPC is $\frac{|M_u| \times |M_v| \times msize}{B}$. The peak aggregate throughput of AAPC is bounded by

$$\frac{|M| \times (|M| - 1) \times \ msize}{\frac{|M_u| \times |M_v| \times msize}{B}} = \frac{|M| \times (|M| - 1) \times \ B}{M_u \times M_v}$$

In general networks, this peak aggregate throughput may not be achieved due to node and link congestion. However, as will be shown later, for the tree topology, this physical limit can be approached through message scheduling.

3.2.2 AAPC Message Scheduling

In the following, an algorithm that constructs phases for AAPC is presented. The phases conform to the following constraints, which are sufficient to guarantee optimality: (1) no contention within each phase; (2) every message in AAPC appears exactly once in the phases; and (3) the total number of phases is equal to the load of AAPC on a given topology. If phases that satisfy these constraints can be carried out without inter-phase interferences, the peak aggregate throughput is achieved.

The scheduling algorithm has three components. The first component identifies the *root* of the system. For a graph $G = (S \cup M, E)$, the *root* is a switch that satisfies two conditions: (1) it is connected to a bottleneck edge; and (2) the number of machines in each of the subtrees connecting to the root is less than or equal to $\frac{|M|}{2}$. The second component performs global message scheduling that determines the phases when messages between two subtrees are carried out. Finally, the third component performs global and local message assignment, which decides the final scheduling of local and global messages.

3.2.2.1 Identifying the Root

Let the tree be $G = (S \cup M, E)$. The process to find a root in the network is as follows. Let link $L = (u, v)$ be one of the bottleneck links. Link L partitions the tree into two subtrees G_u and G_v. The load of link L is thus, $|M_u| \times |M_v| = (|M| - M_v) \times |M_v|$. Assume that $|M_u| \geq |M_v|$. If in G_u, node u has more than one branch containing machines, then node u is the root. Otherwise, node u should have exactly one branch that contains machines (obviously this branch may also have switches). Let the branch connect to node u through link (u_1, u). Clearly, link (u_1, u) is also a bottleneck link since all machines in G_u are in G_{u_1}. Thus, the process can be repeated for link (u_1, u). This process can be repeated n times and n bottleneck links (u_n, u_{n-1}), (u_{n-1}, u_{n-2}), ..., (u_1, u), are considered until the node u_n has more than one branch containing machines in G_{u_n}. Then, u_n is the root. Node u_n should have a nodal degree larger than 2 in G.

Lemma 1: Using the above process to find the root, each subtree of the root contains at most $\frac{|M|}{2}$ machines.

Proof: Using the process described above, a root u_n and the connected bottleneck link (u_n, u_{n-1}) are identified. Let $G_{u_n} = (S_{u_n} \cup M_{u_n}, E_{u_n})$ and $G_{u_{n-1}} = (S_{u_{n-1}} \cup M_{u_{n-1}}, E_{u_{n-1}})$ be the two connected components after link (u_n, u_{n-1}) is removed from G. We have $|M_{u_n}| \geq |M_{u_{n-1}}|$, which implies $|M_{u_{n-1}}| \leq \frac{|M|}{2}$. The load on the bottleneck link (u_n, u_{n-1}) is $|M_{u_n}| \times |M_{u_{n-1}}|$. Let node w be any node that connects to node u_n in G_{u_n} and $G_w = (S_w \cup M_w, E_w)$ be the corresponding subtree. We have $\frac{|M|}{2} \geq |M_{u_{n-1}}| \geq |M_w|$ [Note: if $|M_{u_{n-1}}| < |M_w|$, the load on link (u_n, w) is greater than the load on link (u_n, u_{n-1}) $(|M_w| \times (|M| - |M_w|) > |M_{u_{n-1}}| \times (|M| - |M_{u_{n-1}}|))$, which contradicts the fact that (u_n, u_{n-1}) is a bottleneck link]. Hence, each subtree of the root contains at most $\frac{|M|}{2}$ machines. □

In Figure 3.2, link $(s0, s1)$ is bottleneck link. Both nodes $s0$ and $s1$ can be the root. Assume that $s1$ is selected as the root. It is connected with three subtrees t_{s0} that contains three machines $n0$, $n1$, and $n2$, t_{s3} that contains two machines $n3$ and $n4$, and t_{n5} that contains one machine $n5$.

3.2.2.2 Global Message Scheduling

Let the root connect to k subtrees, t_0, t_1, ..., t_{k-1}, with $|M_0|, |M_1|, ..., and |M_{k-1}|$ machines respectively. Figure 3.3 shows the two-level view of the network. Only global messages

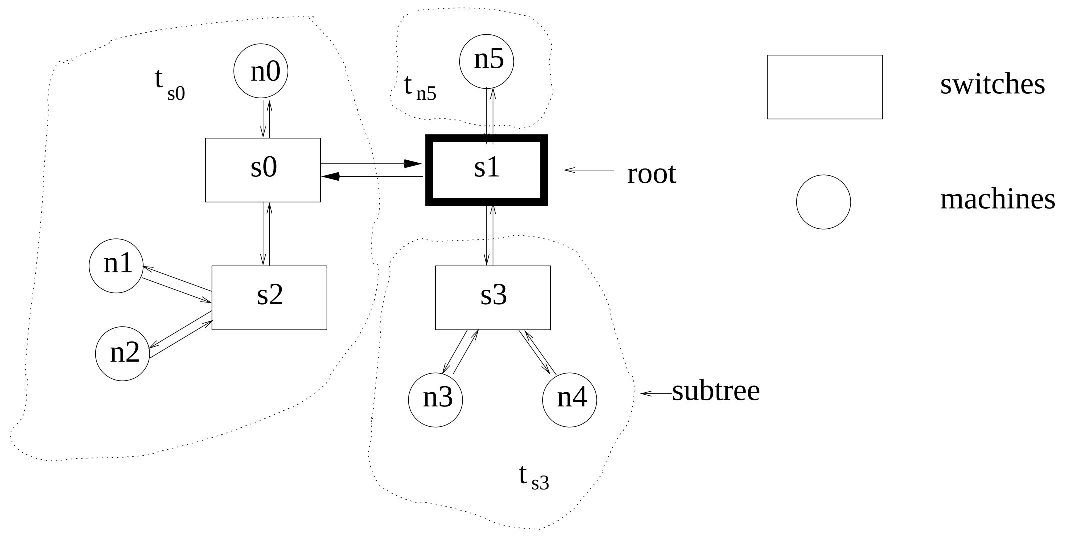


Figure 3.2: Identifying the root in an example cluster

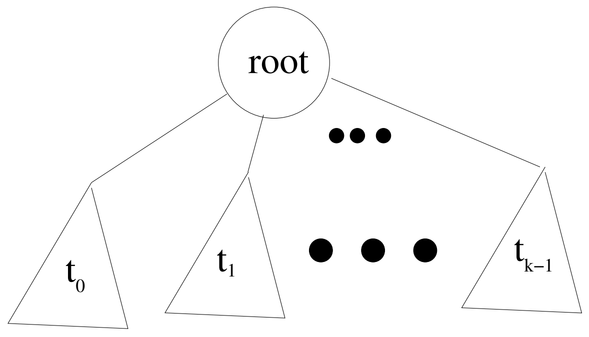


Figure 3.3: A two level view of the network

use the links between the subtrees and the root. Local messages only use links within a subtree. Let us assume that $|M_0| \geq |M_1| \geq ... \geq |M_{k-1}|$. Thus, the load of AAPC is $|M_0| \times (|M_1| + |M_2| + ... + |M_{k-1}|) = |M_0| \times (|M| - |M_0|)$, and we must schedule both local and global messages in $|M_0| \times (|M| - |M_0|)$ phases while maintaining contention-free phases. This is done in two steps. First, phases are allocated for global messages where messages from one subtree to another subtree are treated as groups. Second, individual global and local messages are assigned to particular phases.

The notation $t_i \rightarrow t_j$ will be used to represent either a message from a machine in subtree t_i to a machine in subtree t_j or general messages from subtree t_i to subtree t_j. The global

Table 3.1: Phases from ring scheduling

Phase 0	Phase 1	...	Phase $k-2$
$t_0 \rightarrow t_1$	$t_0 \rightarrow t_2$	...	$t_0 \rightarrow t_{k-1}$
$t_1 \rightarrow t_2$	$t_1 \rightarrow t_3$	...	$t_1 \rightarrow t_0$
...	...	...	...
$t_{k-2} \rightarrow t_{k-1}$	$t_{k-2} \rightarrow t_0$	...	$t_{k-2} \rightarrow t_{k-3}$
$t_{k-1} \rightarrow t_0$	$t_{k-1} \rightarrow t_1$	...	$t_{k-1} \rightarrow t_{k-2}$

message scheduling decides phases for messages in $t_i \rightarrow t_j$. Let us first consider a simple case where $|M_0| = |M_1| = ... = |M_{k-1}| = 1$. In this case, there is $|M_i| \times |M_j| = 1$ message in $t_i \rightarrow t_j$. A ring scheduling algorithm [77, 73] can be used to schedule the messages in $1 \times (k-1) = k-1$ phases. In the ring scheduling, $t_i \rightarrow t_j$ is scheduled at phase $j-i-1$ if $j > i$ and phase $(k-1)-(i-j)$ if $i > j$. The ring scheduling produces $k-1$ phases as shown in Table 3.1.

When scheduling messages with any number of machines in a subtree, all messages from one subtree to another are grouped into consecutive phases. The total number of messages from t_i to t_j is $|M_i| \times |M_j|$. The ring scheduling is extended to allocate phases for groups of messages. In the extended ring scheduling, for subtree t_i, the messages to other subtrees follow the same order as in the ring scheduling. For example, for t_1, messages in $t_1 \rightarrow t_2$ happen before messages in $t_1 \rightarrow t_3$, messages in $t_1 \rightarrow t_3$ happen before messages in $t_1 \rightarrow t_4$, and so on. Specifically, the phases are allocated as follows. Note that messages in $t_i \rightarrow t_j$ occupy $|M_i| \times |M_j|$ consecutive phases.

- When $j > i$, messages in $t_i \rightarrow t_j$ start at phase $p = |M_i| \times \sum_{k=i+1}^{j-1} |M_k|$. If $i+1 > j-1$, $p = 0$.
- When $i > j$, messages in $t_i \rightarrow t_j$ start at phase $p = |M_0| \times (|M| - |M_0|) - (|M_j| \times \sum_{k=j+1}^{i} |M_k|)$.

Lemma 2: Using the extended ring scheduling described above, the resulting phases have the following two properties: (1) the number of phases is $|M_0| \times (|M| - |M_0|)$; and (2) in each phase, global messages do not have contention on links connecting subtrees to the root.
Proof: When $j > i$, messages in $t_i \rightarrow t_j$ start at phase $|M_i| * (|M_{i+1}| + |M_{i+2}| + ... + |M_{j-1}|)$

and end at phase $|M_i| * (|M_{i+1}| + |M_{i+2}| + ... + |M_{j-1}| + |M_j|) - 1 < |M_0| * (|M_1| + ... + |M_{k-1}|) = |M_0| * (|M| - |M_0|)$. When $i > j$, messages in $t_i \rightarrow t_j$ start at phase $|M_0| * (|M| - |M_0|) - (|M_i| + |M_{i-1}| + ... + |M_{j+1}|) * |M_j|$ and end at phase $|M_0| * (|M| - |M_0|) - (|M_i| + |M_{i-1}| + ... + |M_{j+1}|) * |M_j| + |M_i| * |M_j| - 1 < |M_0| * (|M| - |M_0|)$. Thus, the number of phases is less than or equal to $|M_0| * (|M| - |M_0|)$. Note the phase count starts at phase 0. Messages in $t_0 \rightarrow t_{k-1}$ start at phase $|M_0| * (|M_1| + |M_2| + ... + |M_{k-2}|)$ and end at phase $|M_0| * (|M_1| + |M_2| + ... + |M_{k-2}|) + |M_0| * |M_{k-1}| - 1 = |M_0| * (|M| - |M_0|) - 1$. Thus, the number of phases is exactly $|M_0| * (|M| - |M_0|)$. Examining the starting and ending phases for messages in $t_i \rightarrow t_j$, it can be shown that phases for $t_i \rightarrow t_j$, $j \neq i$, do not overlap and that phases for $t_j \rightarrow t_i$, $j \neq i$, do not overlap. Thus, at each phase, at most one node in a subtree is sending and at most one node in a subtree is receiving. As a result, the two edges of the link connecting a subtree to the root will be used at most once in each phase. Hence, in each phase, global messages do not have contention on links connecting subtrees to the root.□

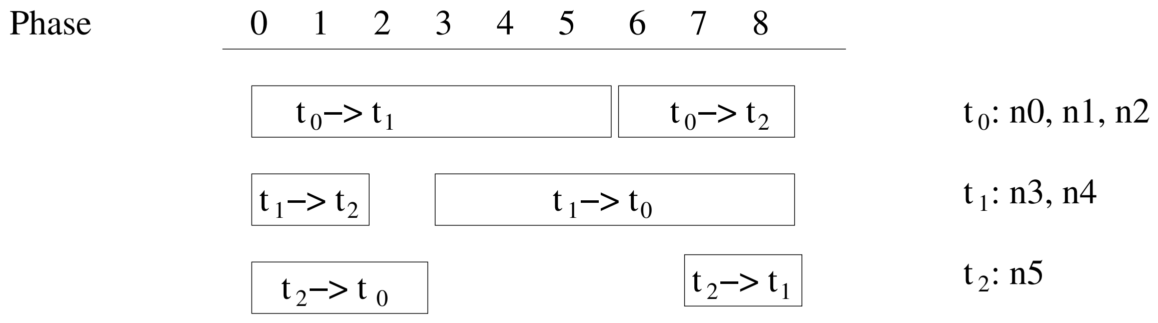


Figure 3.4: Global message scheduling for the example in Figure 3.2

Figure 3.4 shows the scheduling of global messages for the example shown in Figure 3.2. In this figure, there are three subtrees: $t_0 = t_{s0}$ with $|M_0| = 3$, $t_1 = t_{s3}$ with $|M_1| = 2$, and $t_2 = t_{n5}$ with $|M_2| = 1$. Following the above equations, messages in $t_1 \rightarrow t_2$ start at $p = 0$, messages in $t_0 \rightarrow t_2$ start at $p = 6$, and messages in $t_2 \rightarrow t_0$ start at $p = 0$. Figure 3.4 also shows that some subtrees are idle at some phases. For example, subtree t_1 does not have a sending machine in phase 2.

3.2.2.3 Global and Local Message Assignment

Let the root connect to k subtrees, t_0, t_1, ..., t_{k-1}, with $|M_0|$, $|M_1|$, ..., $|M_{k-1}|$ machines, respectively. Assume $|M_0| \geq |M_1| \geq ... \geq |M_{k-1}|$. As shown previously, global messages are scheduled in $|M_0| \times (|M| - |M_0|)$ phases. Consider subtree t_i, the total number of local messages in t_i is $|M_i| \times (|M_i| - 1)$, which is less than the total number of phases. Thus, if in each phase, one local message in each subtree can be scheduled without contention with the global messages, all messages in AAPC can be scheduled in $|M_0| \times (|M| - |M_0|)$ phases. The contention free scheduling of global and local messages is based on the following lemma.

Lemma 3: Let $G = (S \cup M, E)$ be a tree and $x \neq y \neq z \in S \cup M$, $path(x, y) \cap path(y, z) = \phi$.

Proof: Assume that $path(x, y) \cap path(y, z) \neq \phi$. There exists an edge (u, v) that belongs to both $path(x, y)$ and $path(y, z)$. As a result, the composition of the partial path $path(y, u) \subseteq path(y, z)$ and $path(u, y) \subseteq path(x, y)$ forms a non-trivial loop: edge (u, v) is in the loop while edge (v, u) is not. This contradicts the assumption that G is a tree. □

Lemma 4: Using the global message scheduling scheme, at each phase, the global messages do not have contention.

Proof: Let the root connect to subtrees t_0, t_1, ..., t_{k-1}. From Lemma 2, at each phase, there is no contention in the link connecting a subtree to the root. Also, there is no contention when there is only one global message in a subtree in a phase. Thus, the only case when global messages may have contention inside a subtree is when there are two global messages involving nodes in a subtree in a phase: one global message, $x \rightarrow o_1$, is sent into the subtree and the other one, $o_2 \rightarrow y$, is sent out from the subtree ($x, y \in M_i$; o_1 and o_2 are in other subtrees). The sub-path for $x \rightarrow o_1$ inside t_i is equal to $path(x, root)$ and the sub-path for $o_2 \rightarrow y$ is equal to $path(root, y)$. From Lemma 3, these two paths do not have contention inside t_i. □

The contention free scheduling of local messages is also based on Lemma 3. Let $u, v \in t_i$ *and* $u \neq v$. From Lemma 3, there are three cases when message $u \rightarrow v$ can be scheduled without contention (with global messages) in a phase: (1) node v is the sender of a global message, and node u is the receiver of a global message; (2) node v is the sender of a global message, and there is no receiving node of a global message in t_i; and (3) node u is the receiver of a global message, and there is no sending node of a global message. Note that by scheduling at most one local message in each subtree, the scheduling algorithm does

Table 3.2: Rotate pattern for realizing $t_i \to t_j$ when $|M_i| = 6$ and $|M_j| = 4$

phase	comm.	phase	comm	phase	comm	phase	comm
0	$t_{i,0} \to t_{j,0}$	6	$t_{i,0} \to t_{j,2}$	12	$t_{i,1} \to t_{j,0}$	18	$t_{i,1} \to t_{j,2}$
1	$t_{i,1} \to t_{j,1}$	7	$t_{i,1} \to t_{j,3}$	13	$t_{i,2} \to t_{j,1}$	19	$t_{i,2} \to t_{j,3}$
2	$t_{i,2} \to t_{j,2}$	8	$t_{i,2} \to t_{j,0}$	14	$t_{i,3} \to t_{j,2}$	20	$t_{i,3} \to t_{j,0}$
3	$t_{i,3} \to t_{j,3}$	9	$t_{i,3} \to t_{j,1}$	15	$t_{i,4} \to t_{j,3}$	21	$t_{i,4} \to t_{j,1}$
4	$t_{i,4} \to t_{j,0}$	10	$t_{i,4} \to t_{j,2}$	16	$t_{i,5} \to t_{j,0}$	22	$t_{i,5} \to t_{j,2}$
5	$t_{i,5} \to t_{j,1}$	11	$t_{i,5} \to t_{j,3}$	17	$t_{i,0} \to t_{j,1}$	23	$t_{i,0} \to t_{j,3}$

not have to consider the specific topologies of the subtrees.

Let us now consider global messages assignment. Let us number the nodes in subtree t_i as $t_{i,0}$, $t_{i,1}$, ..., $t_{i,(|M_i|-1)}$. To realize inter-subtree communication $t_i \to t_j$, $0 \le i \ne j < k$, each node in t_i must communicate with each node in t_j in the allocated $|M_i| \times |M_j|$ phases. The algorithm uses two different methods to realize inter-subtree communications. The first scheme is the *broadcast* scheme. In this scheme, the $|M_i| \times |M_j|$ phases are partitioned into $|M_i|$ rounds with each round having $|M_j|$ phases. In each round, a different node in t_i sends one message to each of the nodes in t_j. This method has the flexibility in selecting the order of the senders in t_i in each round and the order of the receivers in t_j within each round. The following pattern is an example of such scheme:

$t_{i,0} \to t_{j,0}, ..., t_{i,0} \to t_{j,|M_j|-1}$, $t_{i,1} \to t_{j,0}, ...$, $t_{i,1} \to t_{j,|M_j|-1}$, ..., $t_{i,|M_i|-1} \to t_{j,0}, ..., t_{i,|M_i|-1} \to t_{j,|M_j|-1}$.

The second scheme is the *rotate* scheme. Let D be the greatest common divisor of $|M_i|$ and $|M_j|$. Thus, $|M_i| = a \times D$ and $|M_j| = b \times D$. In this scheme, the pattern for receivers is a repetition of M_i times of some fixed sequence that enumerates all nodes in t_j. One example of a fixed sequence is $t_{j,0}, t_{j,1}, ...t_{j,|M_j|-1}$, which results in a receiver pattern of $t_{j,0}, t_{j,1}, ...t_{j,|M_j|-1}, t_{j,0}, t_{j,1}, ...t_{j,|M_j|-1}, ..., t_{j,0}, t_{j,1}, ...\ t_{j,|M_j|-1}$. Unlike the broadcast scheme, in a rotate scheme, the sender pattern is also an enumeration of all nodes in t_i in every $|M_i|$ phases. There is a *base sequence* for the senders, which can be an arbitrary sequence that covers all nodes in t_i. In the scheduling, the base sequence and the "rotated" base sequence are used. Let the base sequence be $t_{i,0}, t_{i,1}, ...t_{i,|M_i|-1}$. The base sequence can be rotated once, which produces the sequence $t_{i,1}, ...t_{i,|M_i|-1}, t_{i,0}$. Sequence $t_{i,2}, ...t_{i,|M_i|-1}, t_{i,0}, t_{i,1}$ is the result of rotating the base sequence twice. The result from rotating the base sequence n

Table 3.3: Mapping between senders and the receivers in Step 2

round 0		round 1		...	round $\|M_0\|-2$		round $\|M_0\|-1$		...
send	recv	send	recv	...	send	recv	send	recv	...
$t_{0,0}$	$t_{0,1}$	$t_{0,0}$	$t_{0,2}$	...	$t_{0,0}$	$t_{0,\|M_0\|-1}$	$t_{0,0}$	$t_{0,0}$	...
$t_{0,1}$	$t_{0,2}$	$t_{0,1}$	$t_{0,3}$	...	$t_{0,1}$	$t_{0,0}$	$t_{0,1}$	$t_{0,1}$	...
...	...	...	...	...	...	...	...	...	...
$t_{0,\|M_0\|-2}$	$t_{0,\|M_0\|-1}$	$t_{0,\|M_0\|-2}$	$t_{0,0}$	...	$t_{0,\|M_0\|-2}$	$t_{0,\|M_0\|-3}$	$t_{0,\|M_0\|-2}$	$t_{0,\|M_0\|-2}$	...
$t_{0,\|M_0\|-1}$	$t_{0,0}$	$t_{0,\|M_0\|-1}$	$t_{0,1}$	...	$t_{0,\|M_0\|-1}$	$t_{0,\|M_0\|-2}$	$t_{0,\|M_0\|-1}$	$t_{0,\|M_0\|-1}$	...

times can be defined similarly. The senders are scheduled as follows. The base sequence is repeated b times for the first $a \times b \times D$ phases. For phases that are multiples of $a \times b \times D$ phases, rotations are then performed to find a new sequence that is repeated b times. It can be shown that all messages in $t_i \rightarrow t_j$ are realized in the rotate scheme.

Table 3.2 shows an example when $|M_i| = 6$ and $|M_j| = 4$. In this case, $a = 3$, $b = 2$, and $D = 2$. The receivers repeat the pattern $t_{j,0}, t_{j,1}, t_{j,2}, t_{j,3}$. The base sequence for the senders is $t_{i,0}, t_{i,1}, t_{i,2}, t_{i,3}, t_{i,4}, t_{i,5}$. This sequence is repeated 2 times. At phase $2 * 3 * 2 = 12$, the senders follow a rotated sequence $t_{i,1}, t_{i,2}, t_{i,3}, t_{i,4}, t_{i,5}, t_{i,0}$ and repeat the pattern 2 times. It can be verified that all messages in $t_i \rightarrow t_j$ are realized.

The following two lemmas illustrate the properties of the broadcast pattern and the rotate pattern.

Lemma 5: In the broadcast pattern that realizes $t_i \rightarrow t_j$, each sender $t_{i,k}$ occupies $|M_j|$ continuous phases.

Proof: Straight-forward from the definition of the broadcast pattern. $\Box$.

Lemma 6: In the rotate pattern that realizes $t_i \rightarrow t_j$, counting from the first phase for messages in $t_i \rightarrow t_j$, each sender in t_i happens once in every $|M_i|$ phases and each receiver in t_j happens once in every $|M_j|$ phases.

Proof: Straight-forward from the definition of the rotate pattern. $\Box$.

Either the broadcast pattern or the rotate pattern can be used to realize messages in $t_i \rightarrow t_j$, $0 \leq i \neq j < k$. The challenge in the scheduling, however, is that all local messages must be embedded in the $|M_0| \times (|M| - |M_0|)$ phases. The scheduling algorithm is shown in Figure 3.5. The algorithm consists of six steps, explained next.

In the first step, messages from t_0 to all other subtrees t_j, $1 \leq j < k$ are scheduled using

```
Input: Results from global message scheduling that identify which phases are used to
        realize $t_i \rightarrow t_j$ for all $0 \le i \ne j < k$
Output: (1) the phase to realize each global message $m_{i,i_1} \rightarrow m_{j,j_1}$,
            $0 \le i_1 < |M_i|$, $0 \le j_1 < |M_j|$, $0 \le i \ne j < k$.
        (2) the phase to realize each local message $m_{i,i_1} \rightarrow m_{i,i_2}$,
            $0 \le i_1 \ne i_2 < |M_i|$, $0 \le i < k$.

Step 1: Assign phases to messages in $t_0 \rightarrow t_j$, $1 \le j < k$.
    1.a: For each $t_0 \rightarrow t_j$, the receivers in $t_j$ are assigned as follows:
         at phase $p$ in the phases for $t_0 \rightarrow t_j$, machine $m_{j,(p-|M_0|\times(|M|-|M_0|))\ mod\ |M_j|}$
          is the receiver.
         /* it can be verified that a sequence enumerating the nodes in $t_j$ is repeated $|M_0|$
            times in phases for $t_0 \rightarrow t_j$. */
    1.b: For each $t_0 \rightarrow t_j$, the senders in $t_0$ are assigned according to the rotate pattern with
          the base sequence $m_{0,0}, m_{0,1}, ..., m_{0,|M_0|-1}$.

Step 2: Assign phases to messages in $t_i \rightarrow t_0$, $1 \le i < k$.
    2.a: Assign the receivers in $t_i \rightarrow t_0$:
         /* Step 1.b organizes the senders in $t_0$ in such a way that every $|M_0|$ phases, all
             nodes in $t_0$ appear as the sender once. $|M_0|$ phases are called a *round* */
         The receiver pattern in $t_i \rightarrow t_0$ is computed based on the sender pattern in $t_0 \rightarrow t_j$
         according to the mapping shown in Table 3.3. Round $r$ has the same mapping as
         round $r\ mod\ |M_0|$.
         /* the mapping ensures that the local messages in $t_0$ can be scheduled */
    2.b: Assign the senders in $t_i$ using the broadcast pattern with order:
          $m_{i,0}$, $m_{i,1}$, ..., $m_{i,|M_i|-1}$.

Step 3: Schedule local messages in $t_0$ in phase 0 to phase $|M_0| \times (|M_0| - 1)$.
        Message $m_{0,i} \rightarrow m_{0,j}$, $0 \le i \ne j < |M_0|$, is scheduled at the phase where $m_{0,i}$ is the
        receiver of a global message and $m_{0,j}$ is the sender of a global message.

Step 4: Assign phases to global messages in $t_i \rightarrow t_j$, $i > j$ and $j \ne 0$. Use the broadcast
        pattern with receivers repeating pattern $m_{j,0}$, $m_{j,1}$, ..., $m_{j,|M_j|-1}$ for each sender $m_{i,k}$
        and senders following the order $m_{i,0}$, $m_{i,1}$, ..., $m_{i,|M_i|-1}$.

Step 5: Schedule local messages in $t_i$, $1 \le i < k$, in phases for $t_i \rightarrow T_{i-1}$.
       /* the last phase for $t_i \rightarrow T_{i-1}$ is phase $|M_0| \times (|M| - |M_0|) - 1$.*/
        Steps 1 through 4 ensure that for each local message $m_{i,i1} \rightarrow m_{i,i2}$, there is a phase
        in the phases for $t_i \rightarrow T_{i-1}$ such that $m_{i,i2}$ is the sender of a global message and either
        $m_{i,i1}$ is a receiver of a global message or no node in $t_i$ is receiving a global message.
        This step schedules $m_{i,i1} \rightarrow m_{i,i2}$ in this phase.

Step 6: Use either the broadcast pattern or the rotate pattern for global messages in $t_i \rightarrow t_j$,
        $i < j$ and $i \ne 0$. Scheduling these messages would not affect scheduling of local messages.
```

Figure 3.5: The global and local message assignment algorithm

the rotate scheme. First, the receivers in $t_0 \rightarrow t_j$ are assigned such that at phase p, node $t_{j,(p-|M_0|\times(|M|-|M_0|)) \ mod \ |M_j|}$ is the receiver. The pattern aligns the receivers with the receivers in $t_i \rightarrow t_j$ when $i > j$. As will be shown in Step 5, this alignment is needed to correctly schedule local messages. The rotate pattern ensures that each node in t_0 appears once as the sender in every $|M_0|$ phases counting from phase 0.

In the second step, messages in $t_i \rightarrow t_0$ are assigned. In this step, phases are partitioned into rounds with each round having $|M_0|$ phases. The main objective of this step is to schedule all local messages in t_0. This is achieved by creating the pattern shown in Table 3.3, which is basically a rotate pattern for $t_0 \rightarrow t_0$. Since in step 1, each node in t_0 appears as a sender in every $|M_0|$ phases, the scheduling of receivers in $t_i \rightarrow t_0$ can directly follow the mapping in Table 3.3. Using this mapping, every node in t_0 appears as a receiver in every $|M_0|$ phases, which facilitates the use of a broadcast pattern to realize messages in $t_i \rightarrow t_0$, $i > 0$. After the receiver pattern is decided, the senders of $t_i \rightarrow t_0$ are determined using the broadcast scheme with the sender order $t_{i,0}$, $t_{i,1}$, ..., $t_{i,|M_i|-1}$.

Step 3 embeds local messages in t_0 in the first $|M_0| \times (|M_0| - 1)$ phases. Note that $|M_0| \times (|M_0| - 1) \leq |M_0| \times (|M| - |M_0|)$. Since the global messages for nodes in t_0 are scheduled according to Table 3.3, for each $t_{0,n} \rightarrow t_{0,m}$, $0 \leq n \neq m < |M_0|$, there exists a phase in the first $|M_0| \times (|M_0| - 1)$ phases such that $t_{0,n}$ is a receiver of a global message while $t_{0,m}$ is a sender of a global message. Thus, all local messages in t_0, $t_{0,n} \rightarrow t_{0,m}$, $0 \leq n \neq m < |M_0|$, can be scheduled in the first $|M_0| \times (|M_0| - 1)$ phases.

In Step 4, global messages in $t_i \rightarrow t_j$, $i > j$ and $j \neq 0$ are assigned using the broadcast scheme as follows: $t_{i,0} \rightarrow t_{j,0}, ..., t_{i,0} \rightarrow t_{j,|M_j|-1}$, $t_{i,1} \rightarrow t_{j,0}, ..., t_{i,1} \rightarrow t_{j,|M_j|-1}$, $t_{i,|M_i|-1} \rightarrow t_{j,0}, ..., t_{i,|M_i|-1} \rightarrow t_{j,|M_j|-1}$.

In Step 5, local messages in subtrees t_i, t_i, $1 \leq i < k$, are scheduled in the phases for $t_i \rightarrow t_{i-1}$. Note that $|M_{i-1}| \geq |M_i|$ and there are $|M_i| \times |M_{i-1}|$ phases for messages in $t_i \rightarrow t_{i-1}$, which is more than the $|M_i| \times (|M_i| - 1)$ phases needed for local messages in t_i. There are some subtle issues in this step. First, all local messages are scheduled before assigning phases to global messages in $t_i \rightarrow t_j$, $1 \leq i < j$. The reason that global messages in $t_i \rightarrow t_j$, $1 \leq i < j$, do not affect the local message scheduling in subtree t_n, $1 \leq n < k$, is that all local messages are scheduled in phases after the first phase for $t_0 \rightarrow t_n$ (since $|M_n| \times |M_{n-1}| \leq |M_0| \times |M_n|$) while phases for $t_i \rightarrow t_j$, $1 \leq i < j$, are all before that phase. Second, let us examine how exactly a communication $t_{i,i_2} \rightarrow t_{i,i_1}$ is scheduled. From Step 4, the receiver in $t_j \rightarrow t_i$, $j > i$, is organized such that, at phase p, $t_{i,(p-|M_0|\times(|M|-|M_0|))\ mod\ |M_i|}$ is the receiver. From Step 1, receivers in $t_0 \rightarrow t_i$ are also aligned such that at phase p, $t_{i,(p-|M_0|\times(|M|-|M_0|))\ mod\ |M_i|}$ is the receiver. Hence, in the phases for $t_i \rightarrow t_{i-1}$, either $t_{i,(p-|M_0|\times(|M|-|M_0|))\ mod\ |M_i|}$ is a receiver of a global message or no node in t_i is receiving a global message. Thus, at all phases in $t_i \rightarrow t_{i-1}$, it can be assumed that the designated receiver is $t_{i,(p-|M_0|\times(|M|-|M_0|))\ mod\ |M_i|}$ at phase p. In other words, at phase p, $t_{i,(p-|M_0|\times(|M|-|M_0|))\ mod\ |M_i|}$ can be scheduled as the sender of a local message. Now, consider the sender pattern in $t_i \rightarrow t_{i-1}$. Since $t_i \rightarrow t_{i-1}$ is scheduled using the broadcast pattern, each t_{i,i_1} is sending in $|M_{i-1}|$ continuous phases. Since the receiving pattern covers every node, $t_{i,i_2} \in t_i$, in every $|M_i|$ continuous phases and $|M_{i-1}| \geq |M_i|$, there exists at least one phase where t_{i,i_1} is sending a global message and t_{i,i_2} is the designated receiver of a global message. Local message $t_{i,i_2} \rightarrow t_{i,i_1}$ is scheduled in this phase.

Finally, since all local messages are scheduled, either the broadcast scheme or rotate scheme can be used to realize messages in $t_i \rightarrow t_j$, $i < j$ and $i \neq 0$.

Theorem 1: The global and local message assignment algorithm in Figure 3.5 produces phases that satisfy the following conditions: (1) all messages in AAPC are realized in $|M_0| \times (|M| - |M_0|)$ phases; and (2) there is no contention within each phase.

Proof: From Lemma 2, all global messages are scheduled in $|M_0| \times (|M| - |M_0|)$ phases. Step 3 in the algorithm indicates that local messages in t_0 are scheduled in $|M_0| \times (|M_0| - 1)$ phases. In Step 5, all local messages in t_i are scheduled in the phases allocated to communications in $t_i \rightarrow t_{i-1}$. Thus, all messages in AAPC are scheduled in $|M_0| \times (|M| - |M_0|)$ phases.

Lemma 4 shows that there is no contention among global messages in each phase. Since local messages in different subtrees cannot have contention and since in one phase, at most one local message in a subtree is scheduled, the contention can only happen between a global message and a local message inside a subtree. Yet, due to local message assignment done in steps 3 and 5 and from Lemma 3, all local messages have no contention with global messages. Thus, there is no contention within a phase. □

Table 3.4 shows the result of the global and local message assignment for the example in Figure 3.2. In this table, it is assumed that $t_{0,0} = n0$, $t_{0,1} = n1$, $t_{0,2} = n2$, $t_{1,0} = n3$, $t_{1,1} = n4$, and $t_{2,0} = n5$. First, the receiver pattern in $t_0 \rightarrow t_1$ and $t_0 \rightarrow t_2$ is determined. For messages in $t_0 \rightarrow t_1$, $t_{1,(p-9)\ mod\ 2}$ is the receiver at phase p, which means the receiver pattern from phase 0 to phase 5 are $t_{1,1}$, $t_{1,0}$, $t_{1,1}$, $t_{1,0}$, $t_{1,1}$, $t_{1,0}$. After that, the rotation

Table 3.4: Results of global and local message assignment for the cluster in Figure 3.2

phase	global messages			local messages		
	$t_0 \rightarrow \{t_1, t_2\}$	$t_1 \rightarrow \{t_2, t_0\}$	$t_2 \rightarrow \{t_0, t_1\}$	t_0	t_1	t_2
0	$t_{0,0} \rightarrow t_{1,1}$	$t_{1,0} \rightarrow t_{2,0}$	$t_{2,0} \rightarrow t_{0,1}$	$t_{0,1} \rightarrow t_{0,0}$		
1	$t_{0,1} \rightarrow t_{1,0}$	$t_{1,1} \rightarrow t_{2,0}$	$t_{2,0} \rightarrow t_{0,2}$	$t_{0,2} \rightarrow t_{0,1}$		
2	$t_{0,2} \rightarrow t_{1,1}$		$t_{2,0} \rightarrow t_{0,0}$	$t_{0,0} \rightarrow t_{0,2}$		
3	$t_{0,0} \rightarrow t_{1,0}$	$t_{1,0} \rightarrow t_{0,2}$		$t_{0,2} \rightarrow t_{0,0}$		
4	$t_{0,1} \rightarrow t_{1,1}$	$t_{1,0} \rightarrow t_{0,0}$		$t_{0,0} \rightarrow t_{0,1}$	$t_{1,1} \rightarrow t_{1,0}$	
5	$t_{0,2} \rightarrow t_{1,0}$	$t_{1,0} \rightarrow t_{0,1}$		$t_{0,1} \rightarrow t_{0,2}$		
6	$t_{0,0} \rightarrow t_{2,0}$	$t_{1,1} \rightarrow t_{0,0}$				
7	$t_{0,1} \rightarrow t_{2,0}$	$t_{1,1} \rightarrow t_{0,1}$	$t_{2,0} \rightarrow t_{1,0}$		$t_{1,0} \rightarrow t_{1,1}$	
8	$t_{0,2} \rightarrow t_{2,0}$	$t_{1,1} \rightarrow t_{0,2}$	$t_{2,0} \rightarrow t_{1,1}$			

pattern is used to realize all messages in $t_0 \rightarrow t_1$ and $t_0 \rightarrow t_2$. The results are shown in the second column in the table. In the second step, messages in $t_1 \rightarrow t_0$ and $t_2 \rightarrow t_0$ are assigned. Messages in $t_2 \rightarrow t_0$ occupy the first round (first three phases). Since the sender pattern in the first round is $t_{0,0}$, $t_{0,1}$, and $t_{0,2}$, according to Table 3.3, the receiver pattern should be $t_{0,1}$, $t_{0,2}$, $t_{0,0}$. The receivers for $t_1 \rightarrow t_0$ are assigned in a similar fashion. After that, the broadcast pattern is used to realize both $t_1 \rightarrow t_0$ and $t_2 \rightarrow t_0$. In Step 3, local messages in t_0 are assigned in the first $3 \times 2 = 6$ phases according to the assignment of the sender and receiver of global messages in each phase. For example, in phase 0, local message $t_{0,1} \rightarrow t_{0,0}$ is scheduled since node $t_{0,0}$ is a sender of a global message and $t_{0,1}$ is a receiver of a global message. Note that the mapping in Table 3.3 ensures that all local messages in t_0 can be scheduled. In Step 4, $t_2 \rightarrow t_1$ is scheduled with a broadcast pattern. In Step 5, local messages in t_1 and t_2 are scheduled. The local messages in t_1 are scheduled in phases for $t_1 \rightarrow t_0$, that is, from phase 3 to phase 8. The alignment of the receivers in $t_0 \rightarrow t_1$ and $t_2 \rightarrow t_1$ ensures that each machine in t_1 appears as the designated receiver in every $|M_1| = 2$ phases starting from the first phase for $t_0 \rightarrow t_1$. Notice that in phase 6, the designated receiver is $t_{1,1}$. In $t_1 \rightarrow t_0$, each node in t_1 is the sender for $|M_0| = 3$ consecutive phases and the receiver pattern in t_1 covers every node in every 2 phases. All local messages in t_1 can be scheduled. In this particular example, message $t_{1,0} \rightarrow t_{1,1}$ is scheduled at phase 7 where $t_{1,0}$ is a (designated) receiver of a global message and $t_{1,1}$ is a sender of a global message, and

$t_{1,1} \rightarrow t_{1,0}$ is scheduled at phase 4. Finally, in Step 6, the broadcast pattern for messages in $t_1 \rightarrow t_2$ is used.

3.2.3 Message Scheduling Based AAPC Implementations

One naive method to achieve contention-free AAPC is to separate the contention-free phases computed by the message scheduling algorithm using barrier synchronizations. In theory, this implementation achieves contention-free communication for AAPC. In practice, there are two major limitations in this implementation. First, the barrier synchronizations would incur substantial synchronization overheads unless special hardware for the barrier operation such as the Purdue PAPERS [13] is available. Second, using barriers to separate all phases may be overly conservative in allowing the data to be injected into the network. Most network systems have some mechanisms such as buffering to resolve contention. Allowing the network system to resolve a limited degree of contention usually results in a better utilization of network resources than resolving contention at the user layer with barriers. Hence, it may be more efficient to use the contention-free phases to *limit contention* instead of to totally eliminate contention. To address the first limitation, other synchronization mechanisms with less overheads such as the pair-wise synchronization can be used to replace the barriers. To address the second limitation, the separation of the communications in different phases may only be partially enforced (or not enforced) instead of being fully enforced. These issues give rise to many variations in how the contention-free AAPC phases can be used to realize AAPC efficiently. Note that synchronization messages can also cause contention. However, such contention is ignored since synchronization messages are small and such contention can usually be resolved by the network system effectively.

In the following, the variations of message scheduling based AAPC schemes that are used to evaluate the proposed message scheduling algorithm are discussed. A scheme is classified as *fully synchronized* when a synchronization mechanism is used to separate each pair of messages (in different phases) that have contention, *partially synchronized* when a synchronization mechanism is only used to limit the potential network contention, or *not synchronized* when no synchronization mechanism is employed. The implementations that are considered include schemes with no synchronizations, fully and partially synchronized schemes with pair-wise synchronizations, and fully and partially synchronized schemes with barrier synchronizations.

3.2.3.1 Implementations with No Synchronizations

The simplest scheme is to use the contention-free phases to order the send and receive operations without introducing any synchronization mechanism. Ordering the messages according to the contention-free phases may reduce the network contention in comparison to other arbitrary ordering of the messages. This scheme is called the *no-sync.* scheme.

For systems with multiple switches, a machine may be idle in some phases. These idle machines may move messages from one phase to an earlier phase in the *no-sync.* scheme, which destroys the contention-free phase structure. Dummy messages can be added so that all machines are busy in all phases, which may improve the chance for maintaining the contention-free phase structure. Ideally, the dummy communications can happen between any two idle machines in a phase. However, allowing dummy communications between an arbitrary pair of machines significantly increases the complexity for scheduling the dummy messages. In the implementation, a simple approach is taken to limit the dummy communications to be within one switch. Specifically, for each idle machine in a phase, the scheme tries to find another machine in the same switch that does not receive or does not send. If such a machine exists, a dummy communication between the two machines is created. If such a machine does not exist, a dummy self-communication (send to self) is inserted in the phase for the idle machine. This scheme is called the *dummy* scheme.

3.2.3.2 Implementations with Pair-wise Synchronizations

The implementations with no synchronization cannot guarantee contention-free communications. With pair-wise synchronizations, the contention-free communications can be maintained by ensuring that two messages that have contention are carried out at different times. There are two ways to perform the pair-wise synchronizations: *sender-based* and *receiver-based*. In the sender-based synchronization, to separate messages $a \rightarrow b$ in phase p and $c \rightarrow d$ in phase q, $p < q$, the synchronization message $a \rightarrow c$ is sent after a sends $a \rightarrow b$, and c sends $c \rightarrow d$ only after it receives the synchronization message. In the receiver-based synchronization, the synchronization message $b \rightarrow c$ is sent after b finishes receiving $a \rightarrow b$, and c sends $c \rightarrow d$ only after it receives the synchronization message. The sender-based scheme is more aggressive in that the synchronization message may be sent before $a \rightarrow b$ completes. Thus, some data in $a \rightarrow b$ may reside in the network when $c \rightarrow d$ starts. The receiver-based scheme may be over-conservative in that the synchronization message is sent only after the data in $a \rightarrow b$ are copied into the application space in b.

The required synchronizations for the fully synchronized scheme are computed as follows. For every communication in a phase, a check is made to see if a synchronization is needed for every other communication at later phases and a dependence graph is built, which is a directed acyclic graph. After deciding all synchronization messages for all communications, redundant synchronizations in the dependence graph are computed and removed. The redundant synchronizations are the ones that can be derived from other synchronizations. For example, assume that message $m1$ must synchronize with message $m2$ and with another message $m3$. If message $m2$ also needs to synchronize with message $m3$, then the synchronization from $m1$ to $m3$ can be removed. Let $|M|$ and $|S|$ be the numbers of machines and switches respectively. The dependence graph contains $O(|M|^2)$ nodes. The complexity to build the graph is $O(|M|^4|S|^2)$ and the complexity to remove redundant synchronizations is $O(|M|^6)$. Since these computations are performed off-line, such complexity is manageable. In code generation, synchronization messages are added for all remaining edges in the dependence graph. This way, the AAPC algorithm maintains a contention-free schedule while minimizing the number of synchronization messages.

In a partially synchronized scheme, the AAPC phases are partitioned into blocks of phases. The number of phases in a block, bs, is a parameter. Block 0 contains phases 0 to $bs - 1$, block 1 contains phases bs to $2 \times bs - 1$, and so on. The partially synchronized schemes use synchronizations to separate messages in different blocks instead of phases. The order of communications within one block is not enforced. The required synchronizations in a partially synchronized scheme are computed by first computing the required synchronizations for the fully synchronized scheme and then removing the synchronizations within each block.

In summary, there are four types of implementations with pair-wise synchronizations. They are named as follows: *sender all* for the fully synchronized scheme with sender-based synchronizations; *sender partial (bs)* for the partially synchronized scheme with sender-based synchronizations and the parameter bs (the number of phases in a block); *receiver all* for the fully synchronized scheme with receiver-based synchronizations; and *receiver partial (bs)* for the partially synchronized scheme with receiver-based synchronizations.

3.2.3.3 Implementations with Barrier Synchronizations

The fully barrier synchronized AAPC scheme is the one with a barrier between each pair of phases. In the partially barrier synchronized scheme, the AAPC phases are partitioned into blocks of phases. The number of phases in a block, *bs*, is a parameter. A barrier is added between each pair of blocks (one barrier every *bs* phases). There are three variations of partially barrier synchronized schemes: no synchronization within each block, sender-based pair-wise synchronization within each block, and receiver-based pair-wise synchronization within each block. These implementations with barriers are named as follows: *barrier all* for the fully synchronized scheme; *barrier partial & none (bs)* for the partially synchronized schemes with no synchronizations within each block; *barrier partial & sender (bs)* for the partially synchronized schemes with *sender all* within each block; *barrier partial & receiver (bs)* for the partially synchronized scheme with *receiver all* within each block.

Table 3.5: Message scheduling based AAPC schemes used in the evaluation

Name (parameter)	description
No-sync.	no synchronization
Dummy	no synchronization with dummy communications for idle machines
Sender all	fully synchronized with sender-based pair-wise synchronizations
Sender partial (bs)	partially synchronized with sender-based pair-wise synchronizations
Receiver all	fully synchronized with receiver-based pair-wise synchronizations
Receiver partial (bs)	partially synchronized with receiver-based pair-wise synchronizations
Barrier all	fully synchronized with barrier synchronizations
Barrier partial & none (bs)	partially synchronized with barrier synchronizations, no synchronization within each block
Barrier partial & sender (bs)	partially synchronized with barrier synchronizations, *sender all* within each block of phases
Barrier partial & receiver (bs)	partially synchronized with barrier synchronizations, *receiver all* within each block of phases
Tuned scheduling based	the best implementation selected from all of the schemes above

3.2.4 Performance Evaluation

For each of the AAPC variations described previously, a routine generator is developed. The generator takes the topology information as input and automatically produces a customized *MPI_Alltoall* routine that employs the particular scheme for the given topology. The automatically generated routines run on MPICH 2-1.0.1 point-to-point primitives. Also, an automatic tuning system (STAGE-MPI which is discussed in the next chapter) is used to select from all of the message scheduling based schemes the best schemes to form a tuned routine for each topology. Practically, the performance of the tuned routines represents the best performance that can be obtained from the proposed message scheduling based implementations. Table 3.5 gives the names and brief descriptions of the schemes used in the evaluation. Note that although the tuning system can theoretically be used to carry out all experiments, it is only used to generate the tuned routines. All experiments are performed by manually executing the algorithms.

The message scheduling based schemes are compared with the original *MPI_Alltoall* routine in LAM/MPI 7.1.1 [44] and a recent improved MPICH 2-1.0.1 [77]. LAM/MPI 7.1.1 and MPICH 2-1.0.1 are compiled with the default setting. Both LAM/MPI and MPICH *MPI_Alltoall* routines are based on point-to-point primitives. Since LAM/MPI and MPICH have different point-to-point implementations, the LAM/MPI algorithm is also ported to MPICH and the performance of the ported routine is reported, which will be referred to as LAM-MPICH. Hence, in the evaluation, message scheduling based implementations are compared with each other and with native LAM/MPI 7.1.1, native MPICH 2-1.0.1, and LAM-MPICH.

The experiments are performed on a 32-node Ethernet switched cluster. The nodes of the cluster are Dell Dimension 2400 with a 2.8GHz P4 processor, 128MB of memory, and 40GB of disk space. All machines run Linux (Fedora) with 2.6.5-1.358 kernel. The Ethernet card in each machine is Broadcom BCM 5705 with the driver from Broadcom. These machines are connected to Dell PowerEdge 2224 100Mbps Ethernet switches.

The code segment used in the performance measurement is shown in Figure 3.6. A barrier operation is performed after each all-to-all operation to ensure that the communications in different invocations do not affect each other. Since only AAPC with reasonably large messages is considered, the overhead introduced by the barrier operation is insignificant. The results reported are the averages of 50 iterations of *MPI_Alltoall* ($ITER\_NUM = 50$) when $msize \leq 256KB$ and 20 iterations when $msize > 256KB$.

The topologies used in the studied are shown in Figure 3.7, two 24-node clusters in Figure 3.7 (a) and Figure 3.7 (b) and two 32-node clusters in Figure 3.7 (c) and Figure 3.7 (d). These topologies are referred to as topologies (1), (2), (3), and (4). The aggregate throughput, which is defined as $\frac{|M|\times(|M|-1)\times msize}{communication\ time}$, is used as the performance metric and

```
for (i=0; i< WARMUP_ITER; i++) MPI_Alltoall(...);
MPI_Barrier(...);
start = MPI_Wtime();
for (count = 0; count < ITER_NUM; count ++){
   MPI_Alltoall(...);
   MPI_Barrier(...);
}
elapsed_time = MPI_Wtime() - start;
```

Figure 3.6: Code segment for measuring the performance of MPI_Alltoall

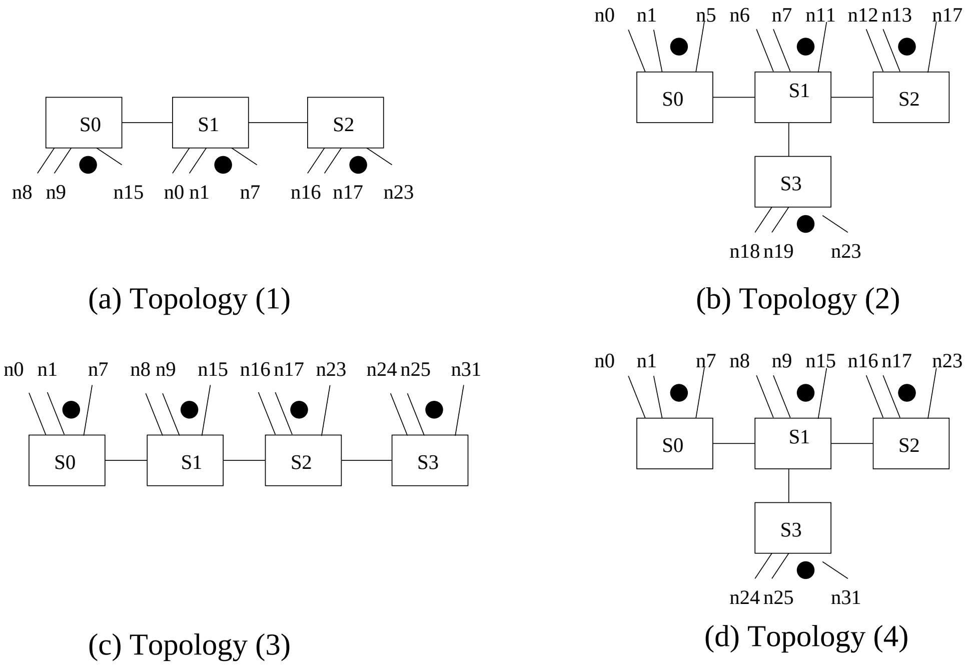


Figure 3.7: Topologies used in the evaluation

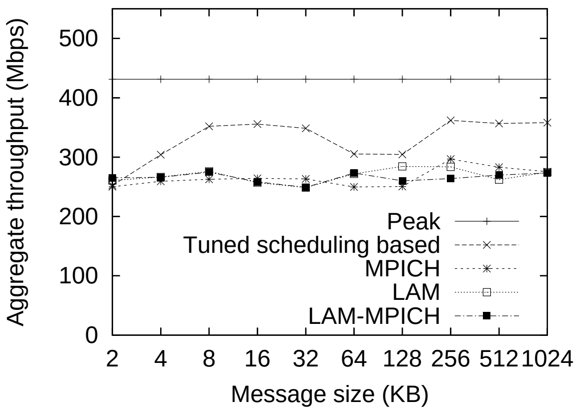
is reported in all experiments.

Figures 3.8 compares the tuned scheduling based implementation with MPICH, LAM, and LAM-MPICH for topologies (1), (2), (3) and (4). In the figures, the theoretical peak aggregate throughput is also shown as a reference. The peak aggregate throughput is obtained using the formula in Section 3.2.1, assuming a link speed of 100Mbps with no additional overheads. The algorithm in LAM/MPI does not perform any scheduling while the improved MPICH performs a limited form of scheduling. Both do not achieve high performance on all topologies since the network contention issue is not fully addressed in the implementations. On the contrary, by introducing proper synchronization into the contention-free AAPC phases, the tuned scheduling based routine consistently achieves (sometimes significantly) higher performance than MPICH, LAM, and LAM-MPICH in the four topologies when the message size is larger than $4KB$. This demonstrates the strength of the message scheduling scheme.

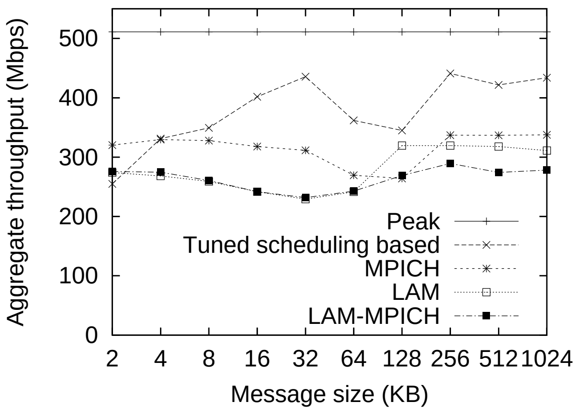
Next, different synchronization mechanisms and different methods to incorporate synchronizations into the contention-free phases in scheduling based AAPC implementations are investigated. The trends in the experimental results for the four topologies are somewhat similar. Thus, for each experiment, only the results for two topologies are reported.

Figure 3.9 compares the receiver-based pair-wise synchronization with the sender-based pair-wise synchronization. When the message size is small, *receiver all* offers better performance. When the message size is large, the sender-based scheme gives better results. With the sender-based pair-wise synchronization, the AAPC scheme injects data into the network aggressively: a message m_e in one phase may not be fully executed (the message may still be in the network system) before the next message m_l that may have contention with m_e starts. Hence, the sender-based scheme allows a limited form of network contention. On the other hand, using the receiver-based pair-wise synchronization, a message m_l that may have contention with an earlier message m_e can start only after the message m_e is received. The results indicate that the limited contention in the sender-based scheme can be resolved by the network system and the sender-based synchronization scheme offers better overall performance when the message size is reasonably large. Since the scheduling based implementations are designed for AAPC with reasonably large messages, the send-based scheme is used for pair-wise synchronization in the rest of the evaluation.

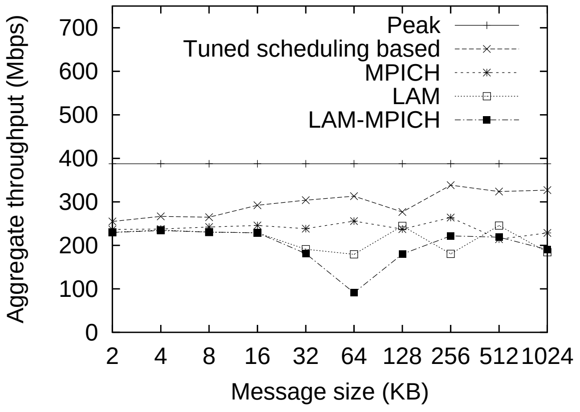
Figure 3.10 compares the performance of message scheduling based AAPC schemes with



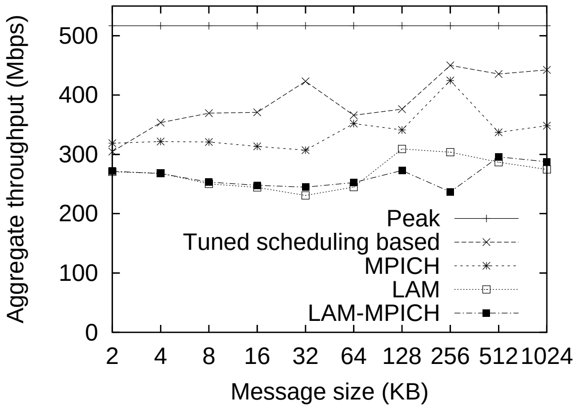
(a) Results for topology (1)



(b) Results for topology (2)



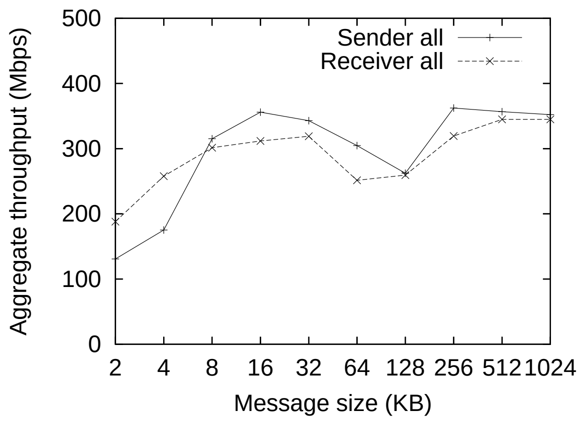
(c) Results for topology (3)



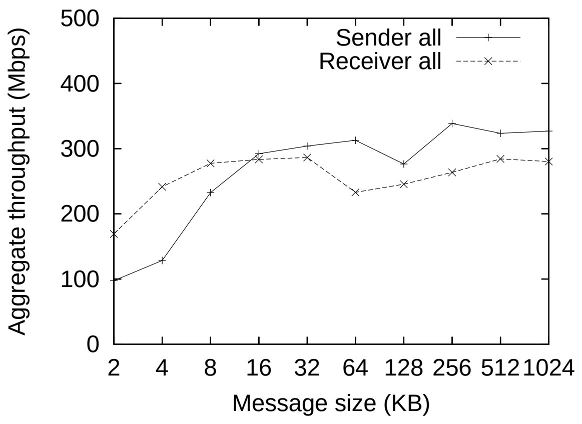
(d) Results for topology (4)

Figure 3.8: The performance of different AAPC implementations

different synchronization mechanisms, including *no-sync.*, *dummy*, *sender all*, and *barrier all*. The aggregate throughput achieved by *no-sync.* and *dummy* is much lower than that achieved by the fully synchronized schemes. Also, adding dummy communications to the idle machines seems to improve the performance over the *no-sync.* scheme in some situations (e.g. topology (2) with $msize = 64KB$) and to degrade the performance in some other situations. Due to the complexity of AAPC, it is unclear whether adding dummy communications is effective in maintaining the phase structure. The fully synchronized scheme with barriers incurs very large overheads when the message size is small. Even when the message size is large, *barrier all* still performs slightly worse than *sender all* in most cases. The $128KB$ case in Figure 3.10 (a) where *barrier all* out-performs *sender all* is an exception. It is

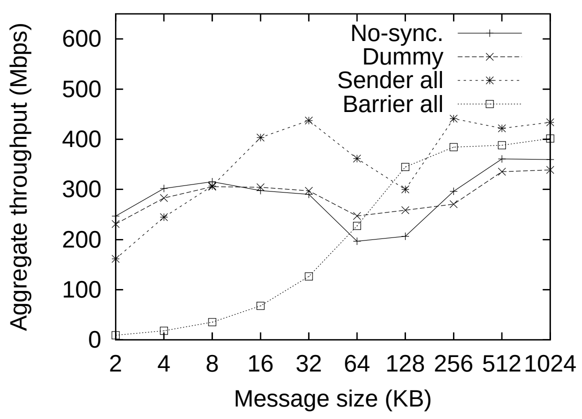


(a) Results for topology (1)

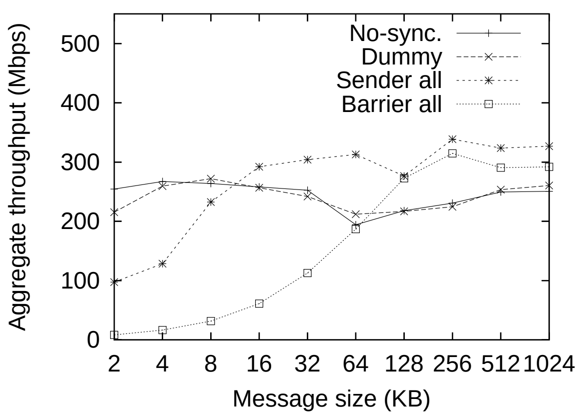


(b) Results for topology (3)

Figure 3.9: Sender-based synchronization versus receiver-based synchronization



(a) Results for topology (2)

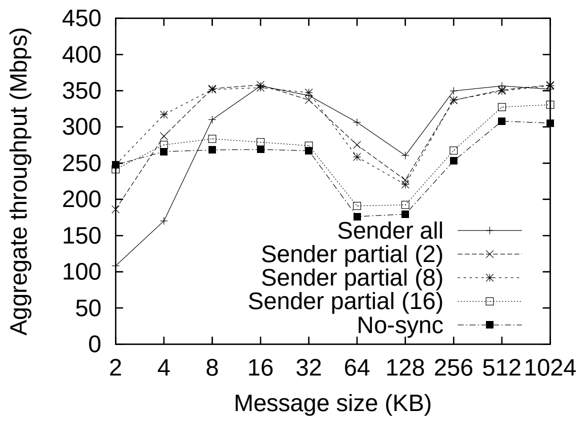


(b) Results for topology (3)

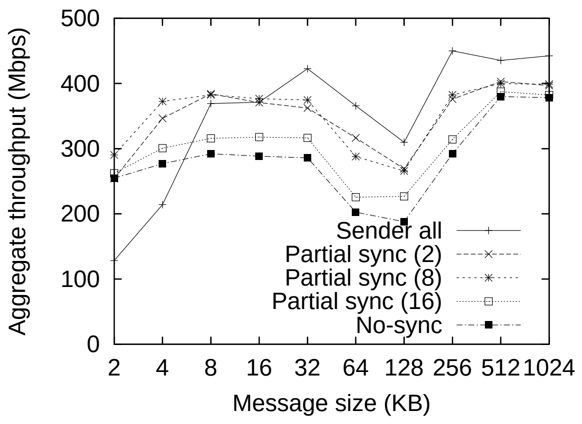
Figure 3.10: Message scheduling based schemes with different synchronization mechanisms

difficult to decide the reason for this case: there are too many factors that can contribute to the performance. Yet, the trend clearly shows that the pair-wise synchronization is more efficient than the barrier synchronization in the implementation of the phased all-to-all communication algorithm.

Figure 3.11 compares the performance of partially synchronized schemes with sender-based pair-wise synchronizations, including *sender partial (2)*, *sender partial (8)*, and *sender partial (16)* with that of *no-sync.* and *sender all*. The trend in the figures is that as the

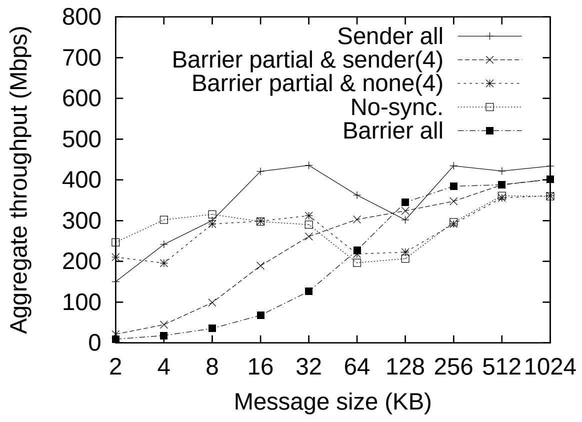


(a) Results for topology (1)

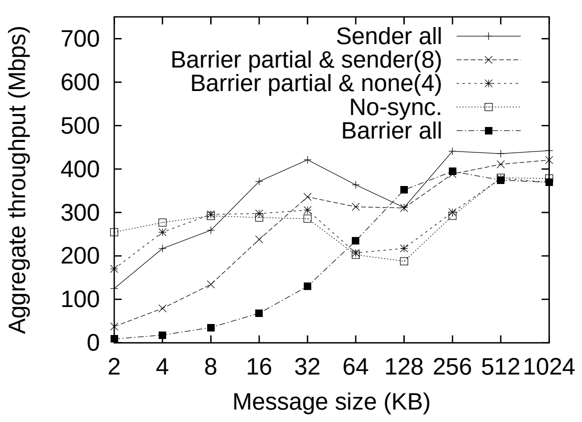


(b) Results for topology (4)

Figure 3.11: Partially synchronized schemes with sender-based pair-wise synchronizations



(a) Results for topology (2)

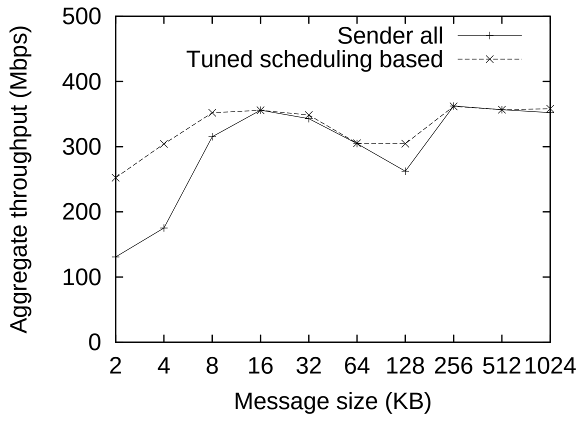


(b) Results for topology (4)

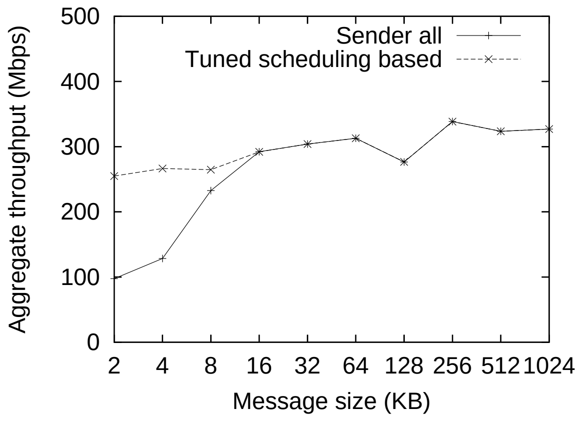
Figure 3.12: Schemes with barrier synchronizations

message size increases, more synchronizations are needed to achieve high performance. The fully synchronized scheme performs the best when the message size is large ($\geq 32KB$). However, the partially synchronized schemes are more efficient for medium sized messages ($2KB$ to $16KB$) than both *no-sync.* and *sender all*.

Figure 3.12 shows the performance of different schemes with barrier synchronizations. When the message size is large, *Barrier partial & none (4)* performs similar to the *no-sync.* scheme. When the message size is small, *Barrier partial & none (4)* incurs



(a) Results for topology (1)

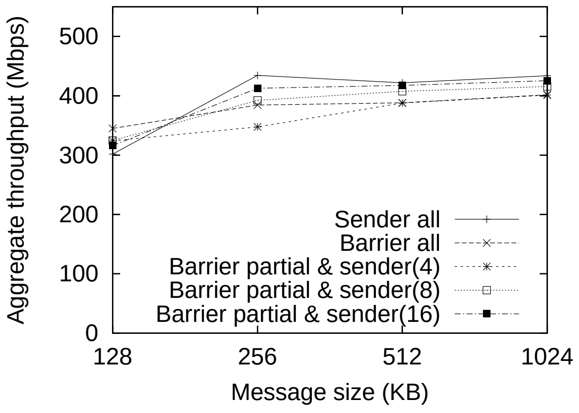


(b) Results for topology (3)

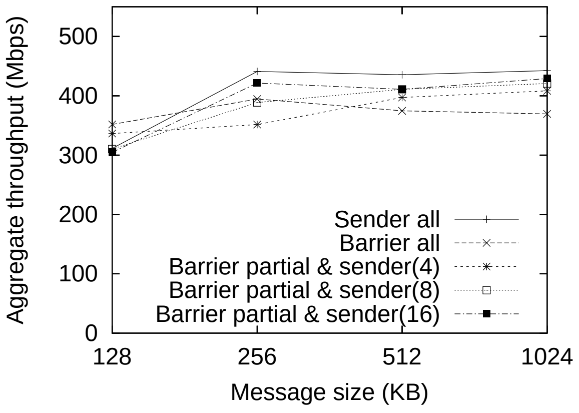
Figure 3.13: Performance of *Sender all* and *tuned scheduling based*

significant overheads. These results indicate that partially synchronized schemes with no synchronizations within each block are not effective. In all experiments, the hybrid barrier and sender-based pair-wise synchronizations never perform better than both *barrier all* and *sender all*, which indicates that such a combination may not be effective. The *sender all* scheme consistently achieves high performance when the message size is reasonably large. Figure 3.13 compares the performance of *sender all* and *tuned scheduling based*. The performance of *sender all* is very close to *tuned scheduling based* when the message size is larger than $16KB$.

Figure 3.14 shows the performance of different synchronization schemes for large messages. As discussed earlier, for large messages, fully synchronized schemes are more effective than partially synchronized schemes. Figure 3.14 shows the results for *sender all*, *barrier all*, *barrier partial & sender (4)*, *barrier partial & sender (8)*, and *barrier partial & sender (16)*. As can be seen from the figure, when the message size is larger than $512KB$, the relative performance of these fully synchronized schemes is quite stable. Ordering the synchronization schemes based on the performance from high to low yields: *sender all*, *barrier partial & sender (16)*, *barrier partial & sender (8)*, and *barrier partial & sender (4)*, and *barrier all*. These results indicate that the sender-based pair-wise synchronization is sufficient even for large messages in the implementation. The heavy weight MPI barrier introduces more overheads without tangible benefits in realizing the phased all-to-all communication.



(a) Results for topology (2)



(b) Results for topology (4)

Figure 3.14: Performance of different fully synchronized schemes for large messages

3.3 All-to-All Broadcast Operation

All–to-all broadcast, also known as all-gather, is another common collective communication operation in high performance computing. In this operation, each node sends the same data to all other nodes in the system. The Message Passing Interface routine that realizes this operation is *MPI_Allgather* [51]. An example all-to-all broadcast on 4 machines is shown in Figure 3.15. Let the number of machines be P. By definition, each node (machine) must receive a total of $(P-1) \times msize$ data from other nodes. Thus, the minimum time to complete the operation is

$$\frac{(P-1) \times msize}{B}.$$

This is the absolute lower bound on the time to complete all–to–all broadcast. Regardless of how the network is connected, no all–to–all broadcast algorithm can have a shorter time.

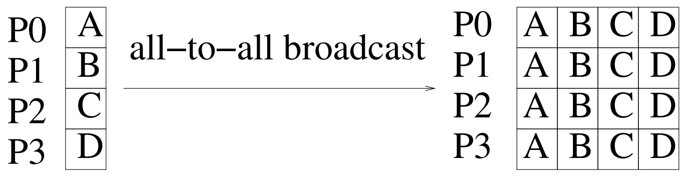


Figure 3.15: All-to-all Broadcast on 4 nodes

This thesis considers all-to-all broadcast on homogeneous clusters connected by either store-and-forward (such as Ethernet) or cut-through (such as Myrinet) switches with arbitrary network topologies (regular or irregular). Let us use the term store-and-forward/cut-through cluster to refer to a cluster with store-and-forward/cut-through switches. In a store-and-forward cluster, the communication time of a message may be significantly affected by the path length of the message, defined as the number of switches a message passes through. In a cut-through cluster, the communication time of a message is virtually independent of the path length of a message.

In this section, algorithms that achieve maximum bandwidth efficiency for all–to–all broadcast on tree topologies are developed. Using these algorithms, all–to–all broadcast on a cut-through cluster with any tree topology has a completion time close to $\frac{(P-1)\times msize}{B}$, the lower bound. In other words, using these algorithms, the reduction in the network connectivity in a tree topology as compared to the system connected by a single switch almost results in no performance degradation for this operation. Since a tree topology can be embedded in most connected networks, it follows that the nearly optimal all-to-all broadcast algorithms can be obtained for most topologies, regular or irregular, by first finding a spanning tree of the network and then applying the proposed algorithms. Note that some routing schemes may prevent a tree from being formed in a connected network. The approach cannot be applied to such systems.

In order to perform all-to-all broadcast efficiently on a store-and-forward cluster, the algorithm must minimize the communication path lengths in addition to achieving maximum bandwidth efficiency. This turns out to be a much harder algorithmic problem. While it cannot be proved formally, this problem is suspected to be NP-complete. However, in this work, the conditions for a store-and-forward cluster with multiple switches to support efficient all-to-all broadcast are identified. In addition, schemes that give optimal solutions for the common cases when each switch is connected to a small number of other switches are developed.

The performance of the algorithms is evaluated using an Ethernet switched cluster with different network topologies. As will be shown later, the performance study confirms that the proposed algorithms achieve nearly optimal performance on clusters with different topologies. Using the proposed algorithms, the performance of all–to–all broadcast on multiple switches is similar to that on a single switch. This result contrasts the results shown in section 3.2

for all–to–all personalized communication, where the network connectivity has significant impacts on the performance. The study also shows that the topology-unaware algorithms used in LAM/MPI[44] and MPICH[52] are not effective on some topologies. Depending on the topology, the proposed algorithms sometimes out-perform the LAM/MPI and MPICH routines to a very large degree (e.g. by a factor of more than 10). In the following, the problem definition is discussed, the proposed schemes to solve the problems are detailed, and the experimental results of the performance study are reported.

3.3.1 Problem Definition

This work focuses on bandwidth efficient all–to–all broadcast schemes, which can be applied when the message size is sufficiently large. Hence, let us assume that the message size, $msize$, is sufficiently large such that communication time is dominated by the bandwidth term. Other communication overheads, such as software startup overheads, are relatively insignificant and are ignored. Let the path length for the message be d. In a cut-through cluster with no network contention, the communication time for a $msize$-byte message is roughly $\frac{msize}{B}$. Note that the communication time in a cut-through cluster is independent of the path length. Let pkt be the packet size in a store-and-forward cluster. The time for transferring a $msize$-byte message in a store-and-forward cluster is roughly $\frac{msize}{B}+(d-1)\times\frac{pkt}{B}$. Depending on the value of $msize$ and pkt, the term $(d-1)\times\frac{pkt}{B}$, introduced by the store-and-forward mechanism, may account for a significant portion of the overall communication time.

In a topology where there are multiple paths between two nodes, the routing issue needs to be considered. However, the major result of this effort is that a tree topology can support the all–to–all broadcast operation as efficiently as any other topology. Since the techniques are developed for the tree topology, where there is only a single path between each pair of nodes and the routing issues do not exist, the focus in the remaining of this section will be on the tree topology and the routing issues will be ignored. The routing issue may need to be considered in the construction of the spanning tree of a general topology.

3.3.1.1 Logical Ring Based All-to-All Broadcast Algorithm

The proposed schemes are based on the logical ring all-to-all broadcast algorithm, which was used for single-switch clusters and two-switch clusters [35, 52]. The algorithm works as follows. Let the cluster contain P machines, numbered as n_0, n_1, ..., n_{P-1}. Let $F : \{0, ..., P-1\} \rightarrow \{0, ..., P-1\}$ be a one-to-one mapping function. Thus, $n_{F(0)}$, $n_{F(1)}$, ..., $n_{F(P-1)}$ is a permutation of n_0, n_1, ..., n_{P-1}. The algorithm works by repeating the following *logical ring pattern* $P-1$ times:

$$n_{F(0)} \rightarrow n_{F(1)} \rightarrow ... \rightarrow n_{F(P-1)} \rightarrow n_{F(0)}.$$

In the first iteration, each machine $n_{F(i)}$, $0 \leq i \leq P-1$, sends its own data to machine $n_{F((i+1)\ mod\ P)}$ and receives data from machine $n_{F((i-1)\ mod\ P)}$. In subsequent iterations, each machine $n_{F(i)}$ forwards what it received in the previous iteration to machine $n_{F((i+1)\ mod\ P)}$ and receives from machine $n_{F((i-1)\ mod\ P)}$. After $P-1$ iterations, all data from all machines reach all machines in the system. Note that in each iteration, each machine must copy the data it receives into the right place of the output buffer.

All logical ring based all-to-all broadcast algorithms operate in the same fashion. The key for such an algorithm to achieve good performance is to find the logical ring pattern that can carry out communications as efficiently as possible. This is the problem to consider.

Let the slowest communication time in the logical ring pattern be $t_{slowest}$. Since the logical ring pattern is repeated $P-1$ times to realize all-to-all broadcast, the total communication time is $(P-1) \times t_{slowest}$. In a cut-through cluster, if there exists a mapping such that the logical ring pattern is contention free, $t_{slowest} \approx \frac{msize}{B}$ and the total time for the all-to-all broadcast operation is $T \approx \frac{(P-1)\times msize}{B}$, which is the theoretical lower bound. Hence, for a cut-through cluster, the challenge is to find a mapping such that the logical ring pattern is contention free. This problem is stated as follows.

Problem 1 (finding a contention free logical ring): Let $G = (S \cup M, E)$ be a tree graph. Let the number of machines in the system be $P = |M|$, and let the machines in the system be numbered as n_0, n_1, ..., n_{P-1}. The problem is to find a one-to-one mapping function $F : \{0, 1, ..., P-1\} \rightarrow \{0, 1, ..., P-1\}$ such that the logical ring pattern $n_{F(0)} \rightarrow n_{F(1)} \rightarrow ... \rightarrow n_{F(p-1)} \rightarrow n_{F(0)}$ is contention free.

For clusters with store-and-forward switches, assuming that the logical ring pattern is contention free and that the longest path length in the pattern is d, $t_{slowest} \approx (d-1)\frac{pkt}{B} + \frac{msize}{B}$, and the total time is $T \approx \frac{(P-1)\times msize}{B} + (d-1) \times (P-1) \times \frac{pkt}{B}$. Hence, to minimize the communication time, the logical ring must (1) be contention free, and (2) have the smallest d, the longest path length in the ring. This problem is stated as follows.

Problem 2 (Finding a contention free logical ring with the smallest maximum path length):

Let $G = (S \cup M, E)$ be a tree graph. Let the number of machines in the system be $P = |M|$, and let the machines in the system be numbered as n_0, n_1, ..., n_{P-1}. The problem is to find a mapping function $F : \{0, 1, ..., P-1\} \rightarrow \{0, 1, ..., P-1\}$ such that (1) the logical ring pattern $n_{F(0)} \rightarrow n_{F(1)} \rightarrow ... \rightarrow n_{F(P-1)} \rightarrow n_{F(0)}$ is contention free, and (2) $\max_{0 \le i \le P-1} \{length(n_{F(i)} \rightarrow n_{F((i+1) \ mod \ P)})\}$ is minimized.

Clearly, Problem 1 is a sub-problem of Problem 2. Unfortunately, only a polynomial time solution for Problem 1 is developed, but not for Problem 2. It is strongly believed that Problem 2 is NP-complete although it cannot be proved formally. Common special cases of Problem 2 when each switch is connected with a small number of switches are considered and a polynomial algorithm that finds the optimal solutions for such cases is developed. Also, the necessary and sufficient conditions is established for a store-and-forward cluster to have a contention-free logical ring with a maximum path length of 2. It must be noted that the topologies in most practical clusters have small diameters. The solution for Problem 1 can be directly applied to such clusters to obtain nearly optimal performance.

3.3.2 Constructing Contention Free Logical Rings

In the following, the polynomial time algorithm for solving Problem 1 is presented. Also, optimal solutions for special cases of Problem 2 are discussed.

3.3.2.1 Problem 1

Let $G = (S \cup M, E)$ be a tree graph. Let the number of machines in the system be $P = |M|$ and the machines in the system be numbered as n_0, n_1, ..., n_{P-1}. This numbering scheme is called global numbering. Let us assume that all switches are intermediate nodes in the tree. Let $G' = (S, E')$ be a subgraph of G that only contains switches and the links between switches. The algorithm, which will be called *Algorithm 1*, determines the mapping for a contention free logical ring pattern in two steps.

- Step 1: Number the switches based on the Depth First Search (DFS) of G'. An example DFS numbering of the switches is shown in Figure 3.16. The switches are denoted as s_0, s_1, ..., $s_{|S|-1}$, where s_i is the ith switch arrived in DFS traversal of G'.

- Step 2: Let the X_i machines connecting to switch s_i, $0 \le i \le |S| - 1$, be numbered as $n_{i,0}$, $n_{i,1}$, ..., n_{i,X_i-1}. This numbering scheme is called local numbering. A one-to-one mapping function (and its reverse function) can be established between the global numbering and local numbering. X_i may be 0 when no machine is connected to s_i. The logical ring $n_{0,0} \rightarrow ... \rightarrow n_{0,X_0-1} \rightarrow n_{1,0} \rightarrow ... \rightarrow n_{1,X_1-1} \rightarrow ...n_{|S|-1,0} \rightarrow ... \rightarrow n_{|S|-1,X_{|S|-1}-1} \rightarrow n_{0,0}$ is contention free (this will be formally proved). The mapping function F for the above logical ring pattern can be obtained using the mapping function from the global numbering to the local numbering.

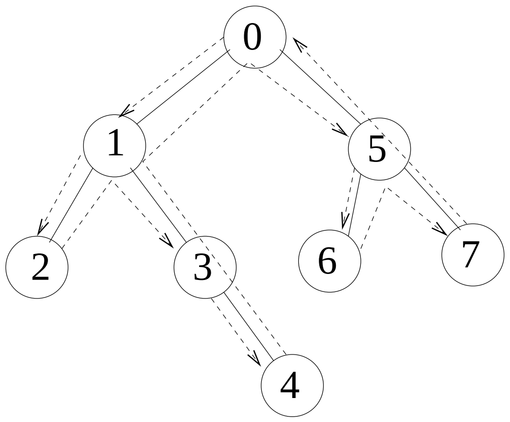


Figure 3.16: DFS numbering

Lemma 7: Let $G' = (S, E')$ be the subgraph of G that contains only switches and links between switches. Let s_0, s_1, ..., $s_{|S|-1}$ be the DFS ordering of the switches, where s_i is the ith switch arrived in DFS traversal of G'. Communications in the following pattern are contention free: $s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_{|S|-1} \rightarrow s_0$.

Proof: Figure 3.16 shows an example DFS numbering of the switches. One can easily see that in this example, pattern $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow ... \rightarrow s_7 \rightarrow s_0$ is contention free. Next, this lemma is formally proved by induction.

Base case: When there is one switch, there is no communication and thus no contention.

Induction case: Assume that the communication pattern in a k-switch system does not have contention. Consider a $(k + 1)$-switch system with switches s_0, s_1, ..., s_k. Removing the last switch s_k from the system, a k-switch system is obtained. The DFS ordering of the k-switch system is exactly the same as the $(k + 1)$-switch system with s_k removed. Hence, from the induction hypothesis, the communication pattern in the k-switch system, that is,

$s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_{k-1} \rightarrow s_0$, does not have contention. Now, let us consider the $(k+1)$-switch system where we need to prove that pattern $s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_k \rightarrow s_0$ does not have contention. The pattern in the $(k+1)$-switch system adds two communications $s_{k-1} \rightarrow s_k$ and $s_k \rightarrow s_0$ to and removes one communication $s_{k-1} \rightarrow s_0$ from the pattern in the k-switch system. Thus, to prove that the pattern in the $(k+1)$-switch system is contention free, we only need to show that communications $s_{k-1} \rightarrow s_k$ and $s_k \rightarrow s_0$ do not introduce contention.

Based on the way DFS operates, switch s_k must be the child of one of the switches along the path from s_0 to s_{k-1}. Hence, there are three cases to be considered: s_k is a child of s_{k-1}, s_k is a child of a switch s'_i along the path from s_0 to s_{k-1} (excluding s_0 and s_{k-1}), and s_k is a child of s_0, . These three cases are depicted in Figure 3.17. The following facts are used in the proof of the three cases.

- Fact (a): The link directly connecting switch s_k does not have contention with all communications in the k-switch system, that is, $s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_{k-1} \rightarrow s_0$. This is because the link is not part of the k-switch system.
- Fact (b): From the induction hypothesis, communication $s_{k-1} \rightarrow s_0$ does not have contention with communications in pattern $s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_{k-1}$.

Now, let us consider the three cases in Figure 3.17.

- Case (1): Switch s_k is a child of s_{k-1}. $s_{k-1} \rightarrow s_k$ does not have contention with any other communications (Fact (a)). $s_k \rightarrow s_0$ is the concatenation of two paths: $s_k \rightarrow s_{k-1}$ and $s_{k-1} \rightarrow s_0$. $s_k \rightarrow s_{k-1}$ does not have contention with all other communications (Fact (a)) and $s_{k-1} \rightarrow s_0$ does not introduce contention (Fact (b)).
- Case (2): Switch s_k is a child of some switch s'_i along the path from s_0 to s_{k-1}. In this case, $s_{k-1} \rightarrow s_k$ is the concatenation of two paths: $s_{k-1} \rightarrow s'_i$ and $s'_i \rightarrow s_k$. $s_{k-1} \rightarrow s'_i$ does not have contention with all other communications since it is a sub-path of $s_{k-1} \rightarrow s_0$ (Fact (b)). Path $s'_i \rightarrow s_k$ does not cause contention (Fact (a)). Similar arguments apply to $s_k \rightarrow s_0$.
- Case (3): Switch s_k is a child of s_0. This follows similar arguments as in Case (1).

Thus, the pattern $s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_{|S|-1} \rightarrow s_0$ is contention free. □

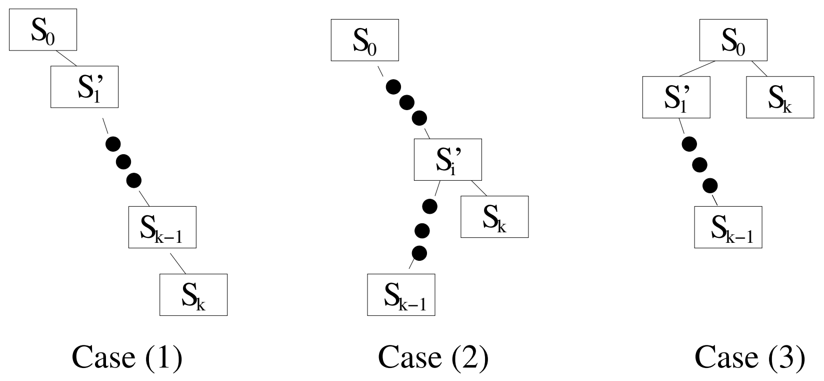


Figure 3.17: Three cases

Theorem 2: The logical ring pattern resulted from *Algorithm 1* is contention free.

Proof: *Algorithm 1* basically obtains the logical ring mapping by (1) grouping all machines connected to a switch together, and (2) ordering the groups of machines based on the DFS order of the switches. To prove that the mapping is contention free, it must be shown that all links between a machine and a switch are contention free and all links between switches are contention free. Since each machine sends and receives exactly once in the logical ring pattern, a link between a machine and a switch is used in both direction exactly once, which indicates that there is no contention on these links. For the links between switches, since the algorithm orders the group of machines (connected to each switch) based on the DFS order, it can be easily shown that the usage of the inter-switch links in the logical ring is exactly the same as the pattern described in Lemma 7. From Lemma 7, there is no contention on the links between switches. □

Using *Algorithm 1*, the contention free logical ring can be found for a tree topology. In networks with an arbitrary topology, this contention free logical ring can be found by first finding a spanning tree and then applying *Algorithm 1*. The two steps may be combined by using the DFS tree to realize the logical ring.

3.3.2.2 Problem 2

To solve Problem 2, a contention free logical ring with the smallest maximum path length must be found. This thesis was not able to either design a polynomial algorithm that exactly solves Problem 2 for all cases or prove this problem to be NP-complete. Thus, this problem is left open. The thesis makes the following two contributions to this problem. First, the sufficient and necessary conditions for a cluster to support a contention free logical ring with a maximum path length of 2 are identified. Note that logical rings with a maximum path length of 1 only exist for clusters connected by a single switch. For clusters with multiple switches, the smallest possible maximum path length in the logical ring is 2 since for each switch, there exists at least one machine that communicates with a machine in another switch. The path length for this communication is at least 2. Hence, this result can be used by a network designer to design a store-and-forward cluster with efficient all-to-all broadcast support. Second, an algorithm that finds optimal solutions is developed for the cases when each switch in the system is connected to a small number of other switches. Here, the term *optimal solutions* means logical rings with the smallest maximum path lengths. A logical ring with a maximum path length of i will be referred to as an i-hop logical ring.

3.3.2.3 Clusters with 2-hop Logical Rings

The sufficient and necessary conditions for a cluster to have a 2-hop logical ring will be formally proved. This result is summarized in the following theorem.

Theorem 3: For a tree graph $G = (S \cup M, E)$, there exists a contention free 2-hop logical ring if and only if the number of machines directly connected to each switch is larger than or equal to the number of switches directly connected to the switch.

Proof: Let us first prove the necessary condition. Assume that there exists a switch, A, that directly connects to more switches than machines. Let us refer to the switches directly connected to A as *A-neighbor* switches. Let all nodes connecting to A through an A-neighbor switch form an A-neighbor subtree. Clearly, the number of A-neighbor subtrees is equal to the number of A-neighbor switches. Under the assumption that all switches are intermediate nodes in the tree topology, each A-neighbor subtree contains at least one machine. Since there are more A-neighbor subtrees than the number of machines attached to A, in the logical ring, at least one machine in an A-neighbor subtree must send a message to a machine in another A-neighbor subtree. The path length of this communication is at least 3 (2 A-neighbor switches plus A). Hence, to have a contention free 2-hop logical ring, each switch must directly connect to at least the same number of machines as the number of switches.

Before the sufficient condition is proved, let us introduce the concept of a logical *array* pattern of a tree (or a subtree), which is rooted at a switch. The term *the logical array of* *a switch* is used to denote the logical array of the tree rooted at the switch. Let the tree (subtree) contain Y machines, n_0, n_1, ..., and n_{Y-1}. Let $F : \{0, ..., Y-1\} \rightarrow \{0, ..., Y-1\}$ be a one-to-one mapping function. The logical array pattern is $n_{F(0)} \rightarrow n_{F(1)} \rightarrow ... \rightarrow n_{F(Y-1)}$. Let us distinguish the first machine, $n_{F(0)}$, and the last machine, $n_{F(Y-1)}$, of the logical array from other machines in the logical array since these two machines must be treated differently. From the definition, it can be seen that the logical array differs from the logical ring by excluding the last communication $n_{F(Y-1)} \rightarrow n_{F(0)}$.

Now, let us consider the sufficient condition. Assume that the number of machines directly connected to each switch is equal to or larger than the number of switches directly connected to the switch. The algorithm performs a post-order traversal of the switches. For each subtree associated with a non-root switch, the algorithm finds the logical array pattern that satisfies the following three conditions: 1) the logical array pattern is contention free, 2) the maximum path length in the pattern is less than or equal to 2, and 3) the first machine, $n_{F(0)}$, and the last machine, $n_{F(Y-1)}$, are directly connected to the root of the subtree. More specifically, the algorithm processes each non-root switch as follows.

- Case (1): The switch does not directly connect to other switches except its parent. Let the switch directly connect to X machines, n_0, ..., n_{X-1}. The array pattern for the switch is $n_0 \rightarrow n_1 \rightarrow ... \rightarrow n_{X-1}$ with the first machine n_0 and the last machine n_{X-1}. It can be verified that the three conditions are met.

- Case (2): The switch directly connects to some machines and some switches other than its parent. Let us use the term "sub-switches" to denote the switches directly connected to the current switch other than its parent. Each sub-switch is the root of a subtree. Since the switches are processed following the post-order traversal order, the logical arrays for all sub-switches have been computed. Let the current switch connect to i sub-switches, denoted as t_0, t_1, ..., t_{i-1}, and j machines, denoted as m_0, m_1, ..., m_{j-1}. We have $j \geq i+1$ since the parent switch does not count in i. For sub-switch t_k, $0 \leq k \leq i-1$, the notations t_k^F, t_k^L, and $t_k^F \rightarrow ... \rightarrow t_k^L$ are used to denote the first machine, the last machine, and the logical array respectively. The logical array for the current switch is $m_0 \rightarrow t_0^F \rightarrow ... \rightarrow t_0^L \rightarrow m_1 \rightarrow t_1^F \rightarrow ... \rightarrow t_1^L \rightarrow m_2 \rightarrow ... \rightarrow m_{i-1} \rightarrow t_{i-1}^F \rightarrow ... \rightarrow t_{i-1}^L \rightarrow m_i \rightarrow m_{i+1} \rightarrow ... \rightarrow m_{j-1}$. This case is depicted in Figure 3.18.

  Now let us examine the three conditions for the logical array of the current switch. It

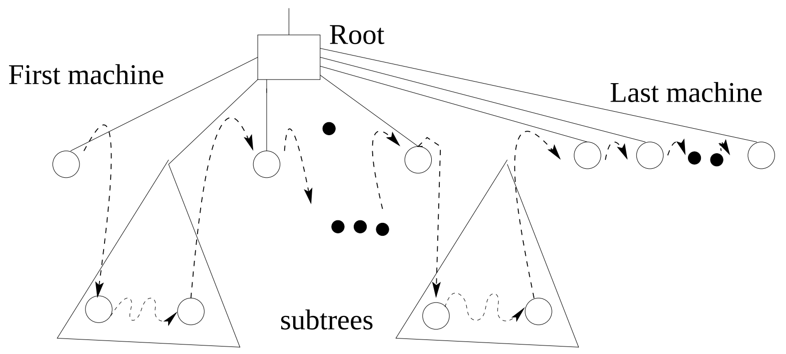


Figure 3.18: Constructing the logical array for an intermediate switch

is obvious that the logical array of the current switch is contention free if the logical arrays of the sub-switches are contention free. The path length for messages $m_k \rightarrow t_k^F$, $0 \leq k \leq i-1$, and messages $t_k^L \rightarrow m_{k+1}$, $0 \leq k \leq i-1$, is exactly 2 since t_k^F and t_k^L are attached to the sub-switch t_k. Since the logical arrays of sub-switches $t_k^F \rightarrow ... \rightarrow t_k^L$, $0 \leq k \leq i-1$, have a maximum path length of 2, the logical array pattern of the current switch has a maximum path length of 2. The first machine m_0 and the last machine m_{j-1} are attached to the current switch. Hence, all three conditions are met.

The processing of root is similar except that the logical ring pattern is constructed instead of the logical array pattern. Let the root directly connect to i top level sub-switches and j machines. When the root does not connect to sub-switches, $i = 0$. Let us denote the i sub-switches as t_0, t_1, ..., t_{i-1} and the j machines as m_0, m_1, ..., m_{j-1}. We have $j \geq i$. For each sub-switch t_k, $0 \leq k \leq i-1$, let us use t_k^F, t_k^L, and $t_k^F \rightarrow ... \rightarrow t_k^L$ to denote the first machine, the last machine, and the logical array respectively. The logical ring pattern for the tree is $m_0 \rightarrow t_0^F \rightarrow ... \rightarrow t_0^L \rightarrow m_1 \rightarrow t_1^F \rightarrow ... \rightarrow t_1^L \rightarrow m_2 \rightarrow ... \rightarrow m_{i-1} \rightarrow t_{i-1}^F \rightarrow ... \rightarrow t_{i-1}^L \rightarrow m_i \rightarrow m_{i+1} \rightarrow ... \rightarrow m_{j-1} \rightarrow m_0$. Note that when $i = j$, t_{i-1}^L sends to m_0 in the logical ring. Following similar arguments as in Case (2), the ring pattern for the root is contention free with a maximum path length less than or equal to 2. Thus, when each switch connects to at least the same number of machines as the number of switches, a contention free 2-hop logical ring can be constructed. □

The proof of the sufficient condition is a constructive algorithm. This algorithm that finds a 2-hop contention free logical ring is called *Algorithm 2*. Figure 3.19 shows the results

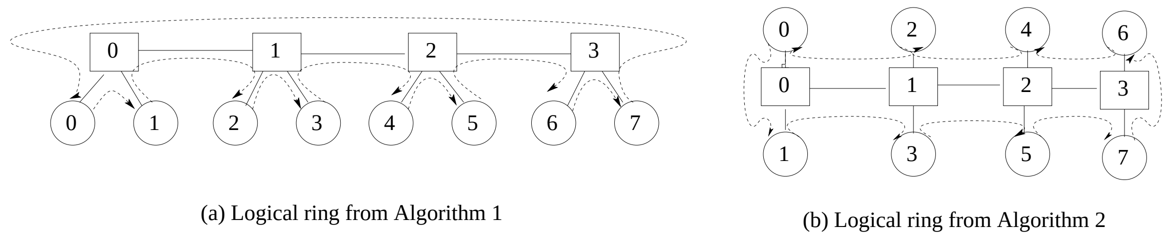

0
1
2
3
0
1
2
3
4
5
6
7
(a) Logical ring from Algorithm 1
(b) Logical ring from Algorithm 2

Figure 3.19: Logical rings from Algorithm 1 and Algorithm 2

of applying *Algorithm 1* and *Algorithm 2* to an 8-machine cluster. As can be seen from the figure, both mappings are contention free. *Algorithm 1* computes a logical ring that has a maximum hop of 4 (from machine 7 to machine 0 in Figure 3.19 (a)) while the logical ring computed using *Algorithm 2* has a maximum hop of 2 as shown in Figure 3.19 (b). For a store-and-forward cluster, using a 2-hop logical ring is expected to perform better than a 4-hop logical ring.

3.3.2.4 Finding Optimal Logical Rings

In the following, an algorithm is described for finding *optimal logical rings*, that is, logical rings with the smallest maximum path lengths. While this algorithm can apply to all topologies, it has a polynomial time complexity only when the number of switches directly connecting to each switch in the system is a small constant. The following lemma provides the foundation for this algorithm.

Lemma 8: Let the P machines in a tree topology $G = (S \cup M, E)$ (rooted at switch R) be numbered as n_0, n_1, ..., n_{P-1}. Let $F : \{0, 1, ..., P-1\} \rightarrow \{0, 1, ..., P-1\}$ be a one-to-one mapping function. The logical ring $n_{F(0)} \rightarrow n_{F(1)} \rightarrow ... \rightarrow n_{F(P-1)} \rightarrow n_{F(0)}$ is contention free if and only if for each subtree that contains X machines, there exists a number $0 \leq i \leq P-1$ such that $n_{F(i)}$, $n_{F(i+1 \ mod \ P)}$, ..., $n_{F(i+X-1 \ mod \ P)}$ are machines in the subtree.

Proof: This lemma states that the necessary and sufficient conditions for a logical ring to be contention free is that all machines in each subtree occupy consecutive positions (the consecutive positions can be wrapped around) in the logical ring. Notice that these conditions apply to **all** subtrees in the system assuming any arbitrary switch as the root. Since only the relative positions of the machines in the logical ring will have an impact on the contention free property of the ring, it can be assumed that the first machine of a top level tree starts at $n_{F(0)}$ in a contention free logical ring without loss of generality.

Let us first prove the necessary condition by contradiction. Assume that the machines in a subtree are not in consecutive positions in the logical ring, there exist at least two numbers i and j such that $n_{F(i)}$ and $n_{F(j)}$ are in the subtree while machines $n_{F(i-1\ mod\ P)}$ and $n_{F(j-1\ mod\ P)}$ are not in the subtree. Since a communication from a machine outside a subtree to a machine inside a subtree must always use the link connecting the subtree to the rest of the tree, communications $n_{F(i-1\ mod\ P)} \rightarrow n_{F(i)}$ and $n_{F(j-1\ mod\ P)} \rightarrow n_{F(j)}$ have contention. This contradicts the assumption that the logical ring is contention free.

To prove the sufficient condition, first, the following claim is proved by induction: Let the P machines in a tree topology rooted at switch R be numbered as n_0, n_1, ..., n_{P-1}. Let $F : \{0, 1, ..., P-1\} \rightarrow \{0, 1, ..., P-1\}$ be a one-to-one mapping function. If the machines in each subtree occupy consecutive positions in the logical array $n_{F(0)} \rightarrow n_{F(1)} \rightarrow ... \rightarrow n_{F(P-1)}$, then communications in $n_{F(0)} \rightarrow n_{F(1)} \rightarrow ... \rightarrow n_{F(P-1)}$, $R \rightarrow n_{F(0)}$, and $n_{F(P-1)} \rightarrow R$ are contention free. In other words, the logical array $n_{F(0)} \rightarrow n_{F(1)} \rightarrow ... \rightarrow n_{F(P-1)}$ is contention free. In addition, the paths from the root to the first machine in the logical array ($n_{F(0)}$) and from last machine in the logical array ($n_{F(P-1)}$) to the root do not have contention with each other and with communications in the logical array.

Base case: It is trivial to show that when there is only a single machine in the system, there is no contention.

Induction case: Consider a tree with n top level subtrees t_0, t_1, ..., t_{n-1}. Since machines in any subtree occupy consecutive positions in the logical array, machines in each subtree t_k, $0 \leq k \leq n-1$ occupy consecutive positions. Let us denote t_k^F, t_k^L, and $T_k = t_k^F \rightarrow ... \rightarrow t_k^L$ be the first machine, the last machine, and the logical array for t_k respectively. Let us denote R_k the root of subtree t_k. Follow the induction hypothesis: communications in $T_k = t_k^F \rightarrow ... \rightarrow t_k^L$, $R_k \rightarrow t_k^F$, and $t_k^L \rightarrow R_k$ are contention free. Let $T_{0'}$, $T_{1'}$, ..., $T_{(n-1)'}$ be a permutation of T_0, T_1, ..., T_{n-1}, where $T_{k'} = t_{k'}^F \rightarrow ... \rightarrow t_{k'}^L$, $0 \leq k \leq n-1$. Since machines in each subtree t_k occupy consecutive positions in the array, the logical array $n_{F(0)} \rightarrow n_{F(1)} \rightarrow ... \rightarrow n_{F(P-1)}$ can be rewritten as $t_{1'}^F \rightarrow ... \rightarrow t_{1'}^L \rightarrow t_{2'}^F \rightarrow ... \rightarrow t_{2'}^L \rightarrow ... \rightarrow t_{(n-1)'}^F \rightarrow ... \rightarrow t_{(n-1)'}^L$. Since all subtrees are disjoint, the contentions in the logical array can be only caused by the inter-subtree communications, $t_{k'}^L \rightarrow t_{(k+1)'}^F$, $0 \leq k \leq n-2$. $t_{k'}^L \rightarrow t_{(k+1)'}^F$ has three components: $t_{k'}^L \rightarrow R_{k'}$, $R_{k'} \rightarrow R \rightarrow R_{(k+1)'}$, and $R_{(k+1)'} \rightarrow t_{(k+1)'}^F$. Since subtree $T_{k'}$ happens once in the logical array, $R_{k'} \rightarrow R \rightarrow R_{(k+1)'}$ will not cause contention. From the induction hypothesis, $t^L_{k'} \rightarrow R_{k'}$ cannot cause contention within subtree $t_{k'}$. Since communications in other subtrees do not use links in $t_{k'}$, $t^L_{k'} \rightarrow R_{k'}$ will not cause contention in the logical array. Similarly, $R_{(k+1)'} \rightarrow t^F_{(k+1)'}$ will not cause contention. Hence, the logical array $n_{F(0)} \rightarrow n_{F(1)} \rightarrow ... \rightarrow n_{F(P-1)}$ (or $t^F_{1'} \rightarrow ... \rightarrow t^L_{1'} \rightarrow t^F_{2'} \rightarrow ... \rightarrow t^L_{2'} \rightarrow ... \rightarrow t^F_{(n-1)'} \rightarrow ... \rightarrow t^L_{(n-1)'}$) is contention free. Similar arguments can be applied to show that communications $R \rightarrow t^F_{1'}$ and $t^L_{(n-1)'} \rightarrow R$ do not have contention between each other and do not have contention with $t^F_{1'} \rightarrow ... \rightarrow t^L_{1'} \rightarrow t^F_{2'} \rightarrow ... \rightarrow t^L_{2'} \rightarrow ... \rightarrow t^F_{(n-1)'} \rightarrow ... \rightarrow t^L_{(n-1)'}$. Thus, communications in $t^F_{1'} \rightarrow ... \rightarrow t^L_{1'} \rightarrow t^F_{2'} \rightarrow ... \rightarrow t^L_{2'} \rightarrow ... \rightarrow t^F_{(n-1)'} \rightarrow ... \rightarrow t^L_{(n-1)'}$, $R \rightarrow t^F_{1'}$, and $t^L_{(n-1)'} \rightarrow R$ are contention free. This finishes the proof of the claim.

It can be shown that $n_{F(P-1)} \rightarrow n_{F(0)}$ is either equal to or a sub-path of $n_{F(P-1)} \rightarrow R \rightarrow n_{F(0)}$. Hence, $n_{F(P-1)} \rightarrow n_{F(0)}$ does not have contention with $n_{F(0)} \rightarrow n_{F(1)} \rightarrow ... \rightarrow n_{F(P-1)}$ and logical ring $n_{F(0)} \rightarrow n_{F(1)} \rightarrow ... \rightarrow n_{F(P-1)} \rightarrow n_{F(0)}$ is contention free. $\square$

Lemma 8 generalizes the results in *Algorithm 1* and *Algorithm 2*, which find two special cases that satisfy the conditions in this lemma. It can be seen that the optimal logical ring is the concatenation of logical arrays for top-level subtrees. The logical arrays for top-level subtrees are the concatenations of the logical arrays for second level subtrees, and so on. The relation between the optimal logical ring and the logical arrays for subtrees is shown in Figure 3.20. This relation motivates solving this problem by reducing the optimal logical ring problem into an optimal logical array problem (optimal logical arrays are the arrays with the smallest maximum path length), which has the optimal substructure property: the optimal logical array for a tree contains the optimal logical arrays for its subtrees. Lemma 8 also indicates that only logical arrays where machines in each subtree occupy consecutive positions need to be considered in order to obtain the optimal logical ring. Hence, the dynamic programming technique can be applied to compute optimal logical arrays for each subtree in a bottom-up fashion.

Let us consider how to reduce the optimal logical ring problem into optimal logical array problems. Let $Opt : \{0, 1, ..., P-1\} \rightarrow \{0, 1, ..., P-1\}$ be a one-to-one mapping function such that $n_{Opt(0)} \rightarrow n_{Opt(1)} \rightarrow ... \rightarrow n_{Opt(P-1)} \rightarrow n_{Opt(0)}$ is an optimal logical ring for a tree topology $G = (S \cup M, E)$ rooted at switch R. Without loss of generality, let us assume that $n_{Opt(0)}$ is the first machine in a top-level subtree. Under the assumption that a switch cannot be a leaf, the root at least has two top-level subtrees. Thus, $n_{Opt(P-1)}$ must be in another top-level subtree and the path $n_{Opt(P-1)} \rightarrow n_{Opt(0)} = n_{Opt(P-1)} \rightarrow R \rightarrow n_{Opt(0)}$.

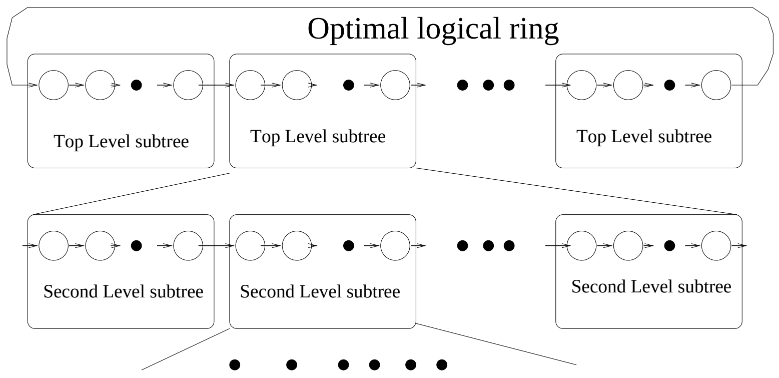


Figure 3.20: The relationship between the optimal logical ring and logical arrays for subtrees

Hence, the optimal logical ring can be considered to have two components: the logical array $n_{Opt(0)} \rightarrow n_{Opt(1)} \rightarrow ... \rightarrow n_{Opt(P-1)}$ and the wrap around link $n_{Opt(P-1)} \rightarrow n_{Opt(0)}$. For a node m, let us use the notation $h(m)$ to denote the height of m, which is defined as the path length from root to m (counting the root as one switch). The height of a node is with respect to a particular subtree (root of the subtree). We have $length(n_{Opt(P-1)} \rightarrow n_{Opt(0)}) = length(n_{Opt(P-1)} \rightarrow R \rightarrow n_{Opt(0)}) = h(n_{Opt(P-1)}) + h(n_{Opt(0)}) - 1$. The length of the wrap around link is a function of the heights of the first machine and the last machine (with respect to the whole tree). Hence, if an optimal logical array $n_{F'(0)} \rightarrow n_{F'(1)} \rightarrow ... \rightarrow n_{F'(P-1)}$ can be found such that $h(n_{F'(0)}) = h(n_{Opt(0)})$ and $h(n_{F'(P-1)}) = h(n_{Opt(P-1)})$, here F' is a one-to-one mapping function, then, the maximum path length of logical ring $n_{F'(0)} \rightarrow n_{F'(1)} \rightarrow ... \rightarrow n_{F'(P-1)} \rightarrow n_{F'(0)}$ is less than or equal to the maximum path length of $n_{Opt(0)} \rightarrow n_{Opt(1)} \rightarrow ... \rightarrow n_{Opt(P-1)} \rightarrow n_{Opt(0)}$, and the logical ring $n_{F'(0)} \rightarrow n_{F'(1)} \rightarrow ... \rightarrow n_{F'(P-1)} \rightarrow n_{F'(0)}$ is also an optimal logical ring. Hence, finding an optimal logical ring can be done by first finding an optimal logical array for each of the possible combinations of $h(n_{Opt(0)})$ and $h(n_{Opt(P-1)})$ and then choosing one that forms a logical ring with the smallest maximum path length. Let the tree height of G be H, the potential values for $h(n_{Opt(0)})$ and $h(n_{Opt(P-1)})$ are in the range of $0..H$.

Next, the algorithm to determine the maximum path length of the optimal logical ring is described. Minor modifications can be made to obtain the actual optimal logical ring. For each node A (a machine or a switch), a two-dimensional table $A.optimal$ is used to store the maximum path length of the optimal logical arrays for the subtree rooted at A. The entry

$A.optimal[i][j]$ stores the maximum path length of the optimal logical array with the height of the first machine being i and the height of the last machine being j. Note that the height is with respect to the subtree rooted at A (the distance from the node to A). Thus, once the *optimal* data structure at the root R, *R.optimal*, is computed, the optimal maximum path length of the logical ring is

$$\min_{i,j}(max(R.optimal[i][j], i + j - 1)).$$

The $R.optimal[i][j]$ is the optimal logical array with the first machine at height i and the last machine at height j, and $i + j - 1$ is the path length of the wrap around link. The term $max(R.optimal[i][j], i + j - 1)$ is the best maximum path length when the ring is formed by having a logical array starting at height i and ending at height j.

Now let us consider how to compute the *A.optimal* data structure at each node A. As in *Algorithm 2*, this data structure is computed in a bottom-up fashion (post-order traversal). For each machine A, $A.optimal[0][0] = 0$ and $A.optimal[i][j] = \infty$, $i \neq 0$ or $j \neq 0$. If A is a switch, all subtrees of A have been processed. Let A have n subtrees t_0, t_2, ..., t_{n-1}. Let us assume that among the n subtrees, k are rooted at switches (each of the subtrees is rooted at a switch). The rest $n - k$ are single-machine subtrees (each of the subtrees contains only a single machine). The algorithm first enumerates all possible different sequences of the n subtrees. Since all machines are the same, switching their positions in the sequence yields the same sequence. Hence, there are at most k-permutation of an n-set (selecting k positions for the k subtrees rooted at switches from the n possible positions in the sequence), that is, $n(n-1)...(n-k+1) = \frac{n!}{(n-k)!} = O(n^k)$, different sequences. Here, factorial function $n! = n \times (n-1) \times ... \times 1$.

For each sequence $seq = t_{0'}t_{1'} \; ... \; t_{(n-1)'}$, the *seq.optimal* data structure is computed for the case when the subtrees are concatenated in the particular order $t_{0'} \rightarrow t_{1'} \rightarrow ... \rightarrow t_{(n-1)'}$. There are three cases. First, if *seq* only has one subtree $t_{0'}$, then $seq.optimal[i][j] = t_{0'}.optimal[i][j]$. Second, if *seq* contains two subtrees $t_{0'}$ and $t_{1'}$, the optimal data structure for the sequence is computed as follows:

$$seq.optimal[i][j] = \min_{k,l}\{max(t_{0'}.optimal[i][k], t_{1'}.optimal[l][j], k + l + 1)\}.$$

To find the optimal logical array $t_{0'} \rightarrow t_{1'}$ that starts at height i and ends at height j, the array in $t_{0'}$ must start at height i and the array in $t_{1'}$ must end at height j. However, the array in $t_{0'}$ can end at any position and the array in $t_{1'}$ can start at any position. For a logical array that is composed of the array in $t_{0'}$ that starts at height i and ends at height

k and the array in $t_{1'}$ that starts at height l and ends at height j, the maximum path length is the maximum of three elements: the maximum path length of the array in $t_{0'}$ ($t_{0'}.optimal[i][k]$), the maximum path length of the array in $t_{1'}$ ($t_{1'}.optimal[l][j]$), and the length of $t_{0'}^L \rightarrow t_{1'}^F = t_{0'}^L \rightarrow A \rightarrow t_{1'}^F$, which is equal to $k+l+1$. This formula takes into account all possible combinations to concatenate the two subtrees. Third, if *seq* contains more than two subtrees, the *seq.optimal* data structure can be obtained by repeatedly applying the concatenation of two logical arrays (the second case).

The optimal data structure for A can be obtained from the optimal data structures for all possible sequences using the following formula:

$$A.optimal[i][j] = \min_{seq\ is\ a\ sequence} \{seq.optimal[i-1][j-1]\}.$$

This formula examines all possible sequences to determine the optimal logical array for a given i and j. Notice that, for a node m, when $h(m) = i$ in subtree rooted at A, $h(m) = i-1$ in the subtrees of A. Intuitively, this algorithm examines all possible cases when all machines in each subtree must be placed in consecutive positions in the logical ring and stores the optimal results at the root of the subtree. By induction, it can be formally shown that this algorithm finds the maximum path length of the optimal logical ring for the tree.

This algorithm, which will be called *Algorithm 3*, operates in a similar fashion to *Algorithm 2*. The difference is that, under the assumptions for *Algorithm 2*, only one optimal logical array for each subtree must be considered to obtain the optimal logical ring for the whole tree. Without those assumptions, many optimal logical arrays for different heights of the first and last machines must be computed and stored. In addition, the process to determine the optimal logical arrays becomes much more complex.

Let us now examine the complexity of this algorithm. Let the number of nodes be $|V|$, the maximum nodal degree be n (n usually equals to the maximum number of ports in a switch), the maximum number of switches directly connecting to one switch be k, the tree height be H. The size of the table to store the optimal logical arrays is $O(H^2)$. The time to concatenate two logical arrays is $O(H^4)$. Since a node can have at most a sequence of size n, computing the optimal data structure for one sequence is $O(nH^4)$. Given that each node can have at most $O(n^k)$ sequences, the time to process each node is then $O(n^k nH^4)$. Thus, the complexity of the whole algorithm is $O(|V|n^{k+1}H^4)$. When k is a small constant, this algorithm has a polynomial time complexity. In practice, while $|V|$ can be a large number (e.g. a cluster of a few thousand nodes), the values of n, k, H are usually small.

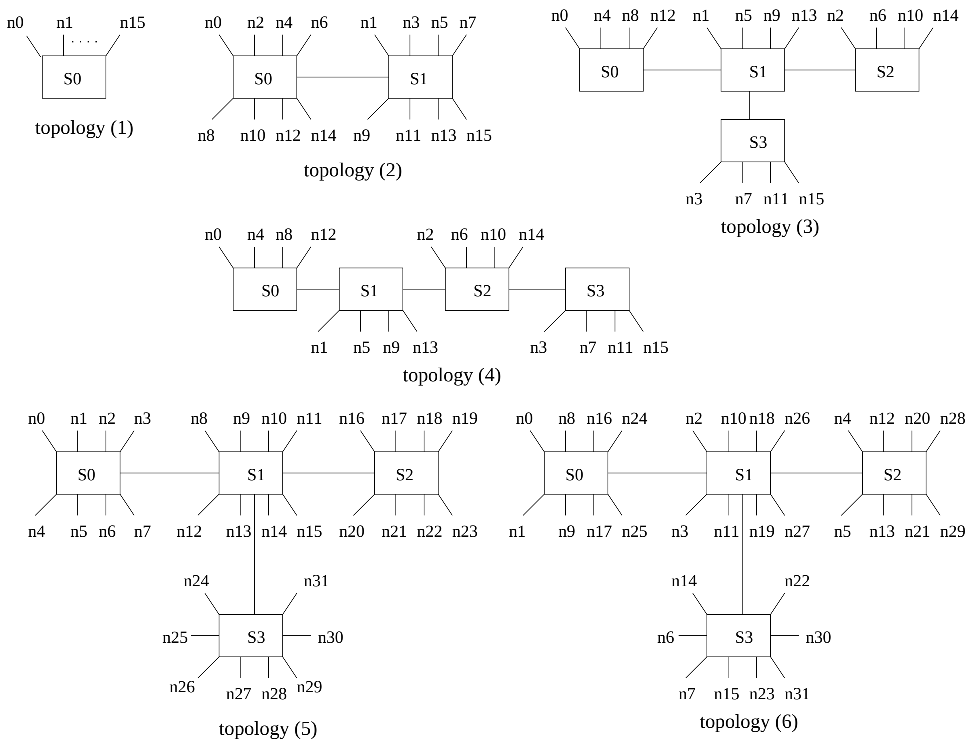
3.3.3 Performance Evaluation

Two approaches are used to evaluate the proposed algorithms. First, an automatic routine generator is developed; it takes the topology information as input and generates, based on the algorithms presented, customized topology-specific all–gather routines. The performance of the generated routines on multiple topologies is compared with that of the all–gather routines in LAM/MPI 7.1.1 [44] and the recently improved MPICH 2-1.0.1 [52]. The experiments are performed on a 32-machine Ethernet switched cluster. Second, the performance of the algorithms is studied on large networks through simulation. In the following, the measurement results on small clusters are presented first and the simulation results are presented second.

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

Figure 3.21: Code segment for measuring MPI_Allgather performance

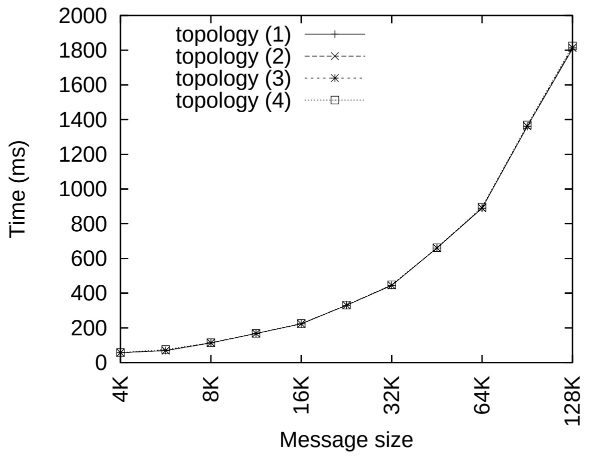
The generated all–gather routines use LAM/MPI 7.1.1 point-to-point primitives and run over LAM/MPI 7.1.1. LR1, LR2, and LR3 are used to denote the routines obtained from *Algorithm 1* (finding a contention free logical ring), *Algorithm 2* (finding a contention free 2-hop logical ring), and *Algorithm 3* (finding an optimal contention free logical ring) respectively. Note that in cases when LR2 can find 2-hop rings, LR3 can also find 2-hop rings. To report fair performance comparison, the MPICH all–gather implementation is ported to LAM/MPI, and MPICH-LAM is used to denote the ported all–gather routine. It was found that, in the latest version of MPICH, the performance of the all–gather operation using native MPICH and MPICH-LAM is very close. In the performance evaluation, LR1, LR2, and LR3 are compared with LAM and MPICH-LAM. To measure the performance of the *MPI_Allgather* routine, the approach similar to Mpptest [32] is used. Figure 3.21 shows an example code segment for measuring the performance. The number of iterations is varied according to the message size: more iterations are used for small message sizes to offset the clock inaccuracy. For the message ranges $4KB - 12KB$, $16KB - 96KB$, and $128KB$, the number of iterations 50, 20, and 10 are used, respectively. The results are the averages of three executions. The *average* time among all machines is used as the performance metric.



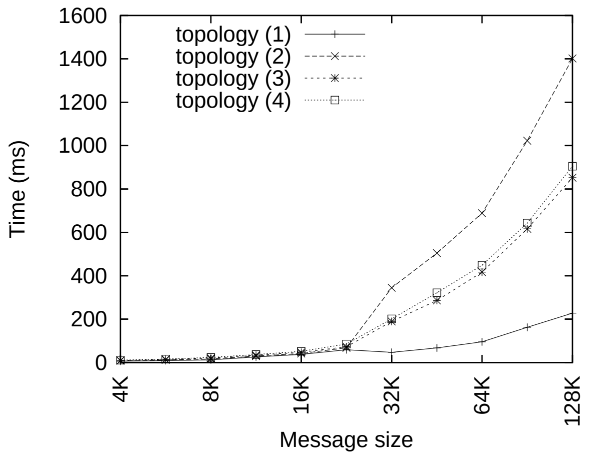
n0 n1 .... n15
S0
topology (1)
n0 n2 n4 n6 n1 n3 n5 n7
S0 S1
n8 n10 n12 n14 n9 n11 n13 n15
topology (2)
n0 n4 n8 n12 n1 n5 n9 n13 n2 n6 n10 n14
S0 S1 S2
S3
n3 n7 n11 n15
topology (3)
n0 n4 n8 n12 n2 n6 n10 n14
S0 S1 S2 S3
n1 n5 n9 n13 n3 n7 n11 n15
topology (4)
n0 n1 n2 n3 n8 n9 n10 n11 n16 n17 n18 n19
S0 S1 S2
n4 n5 n6 n7 n12 n13 n14 n15 n20 n21 n22 n23
n24 n31
n25 S3 n30
n26 n27 n28 n29
topology (5)
n0 n8 n16 n24 n2 n10 n18 n26 n4 n12 n20 n28
S0 S1 S2
n1 n9 n17 n25 n3 n11 n19 n27 n5 n13 n21 n29
n14 n22
n6 S3 n30
n7 n15 n23 n31
topology (6)

Figure 3.22: Topologies used in the experiments

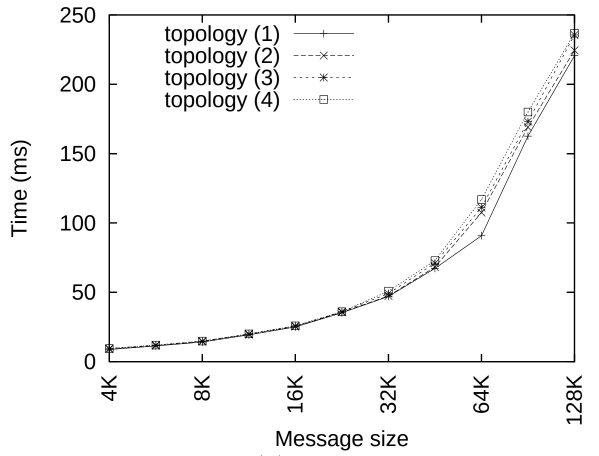
The experiments are performed on a 32-machine Ethernet switched cluster. The machines of the cluster are Dell Dimension 2400 with a 2.8GHz P4 processor, 128MB of memory, and 40GB of disk space. All machines run Linux (Fedora) with 2.6.5-1.358 kernel. The Ethernet card in each machine is Broadcom BCM 5705 with the driver from Broadcom. These machines are connected to Dell PowerConnect 2224 and Dell PowerConnect 2324 100Mbps Ethernet switches. Figure 3.22 shows the topologies used in the experiments. Parts (a) to (d) of the figure represent clusters of 16 machines connected by 1, 2, and 4 switches with different topologies. Parts (e) and (f) show 32-machine clusters of 4 switches, each having 8 machines attached. These two clusters have exactly the same physical topology, but different node assignments. The topologies in the figure are referred to as topology (1), topology (2), topology (3), topology (4), topology (5), and topology (6).



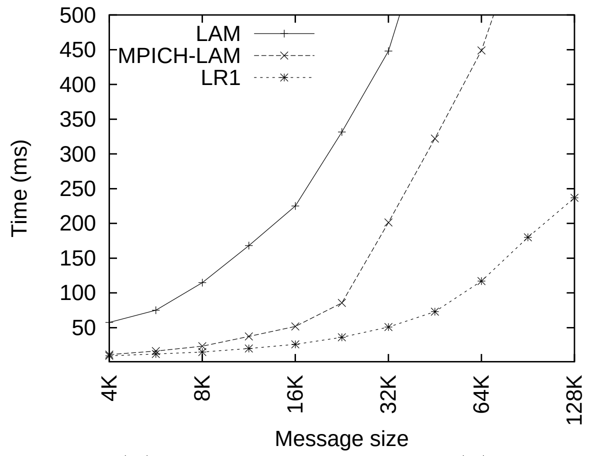
(a) LAM



(b) MPICH-LAM



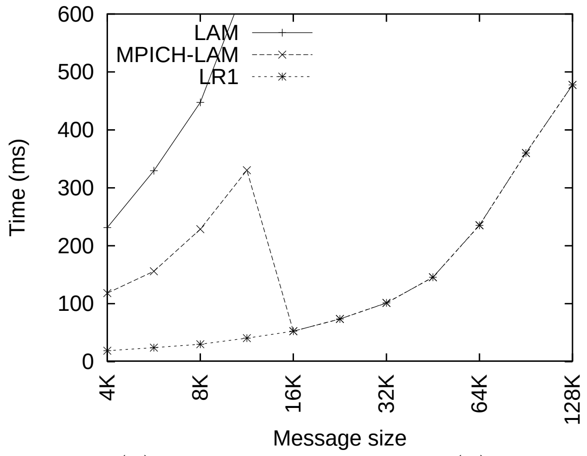
(c) LR1



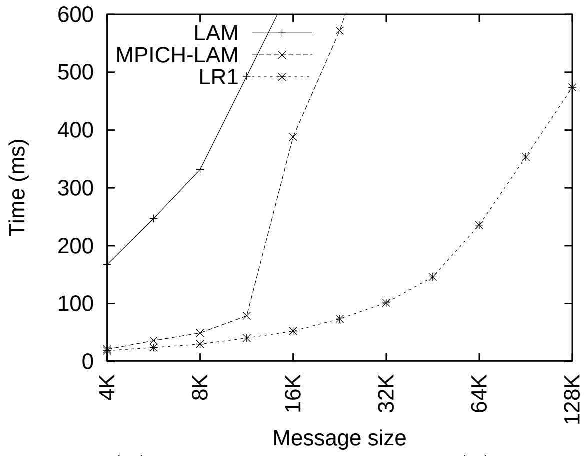
(d) Results for topology (4)

Figure 3.23: Performance of LR1, LAM, and MPICH-LAM on topologies (1) to (4)

Figure 3.23 shows the performance of LAM, MPICH-LAM, and LR1 on topologies (1) to (4). Figure 3.23 (a) shows the performance of LAM. The LAM algorithm has almost the same performance for topologies (1)-(4). The tree-based algorithms used in the LAM all–gather implementation do not exploit all network links and do not create bottleneck



(a) Results for topology (5)



(b) Results for Topology (6)

Figure 3.24: Performance of LR1, LAM, and MPICH-LAM on topologies (5) and (6)

links in all four topologies. However, the network is under-utilized. As can be seen in Figure 3.23 (d), the LAM/MPI routine performs much worse than MPICH and LR1. The results for MPICH are shown in Figure 3.23 (b). MPICH changes the algorithm when $msize = 32KB$. When $msize < 32KB$, MPICH uses the recursive doubling algorithm, which has similar performance for all topologies. Using a topology-unaware logical ring algorithm when $msize \geq 32KB$, MPICH provides very different performance for the four topologies. It performs best on topology (1), where the cluster is connected by a single switch, but significantly worse on topologies (2), (3), and (4), which indicates that the network topology can significantly affect MPICH performance. From Figure 3.23 (c), it can be seen that LR1 achieves very similar performance on all four topologies, which is attributed to the ability of LR1 in finding the contention free logical ring on different topologies. The performance of LR1 on all topologies is similar to the performance of LR1 on the single switch topology (topology (1)). This demonstrates the optimality of LR1 in terms of achieving nearly optimal all-to-all broadcast performance on different topologies. Figure 3.23 (d) compares LR1 with LAM and MPICH on topology (4). It is shown that LR1 performs substantially better than LAM and MPICH.

Figures 3.24 (a) and (b) show the performance results for LAM, MPICH-LAM, and LR1 on topologies (5) and (6) respectively. The extreme poor performance of LAM on

Table 3.6: LR1 vs. LR2 on topology (4)

msize	LR1 topo.(4) (4-hop)	LR2 topo.(4) (2-hop)	LR1 topo.(1) (1-hop)
32KB	50.9ms	48.0ms	47.0ms
48KB	72.9ms	68.4ms	67.2ms
64KB	116.9ms	95.7ms	90.8ms
96KB	180.0ms	172.7ms	162.6ms
128KB	236.8ms	233.7ms	220.6ms

both topologies can be easily seen. As shown in Figure 3.24 (a), the topology-unaware ring algorithm used in MPICH, when $msize \geq 16KB$, achieves nearly optimal performance (same as LR1) for this topology. In this case, the topology-unaware ring algorithm operates exactly the same as LR1. However, with the same physical topology and a different node assignment in topology (6), the topology-unaware algorithm performs much worse than LR1 as shown in Figure 3.24 (b). This again shows that the performance of MPICH depends heavily on the network configuration. Unlike LAM and MPICH, LR1 consistently achieves high performance for different topologies. To illustrate, when the message size is $128KB$, the completion times for LR1, LAM, and MPICH-LAM on topology (6) are 473.7ms, 5346ms, and 3595ms respectively. This means that LR1 achieves a performance that is more than 11 times better than LAM and almost 8 times better than MPICH.

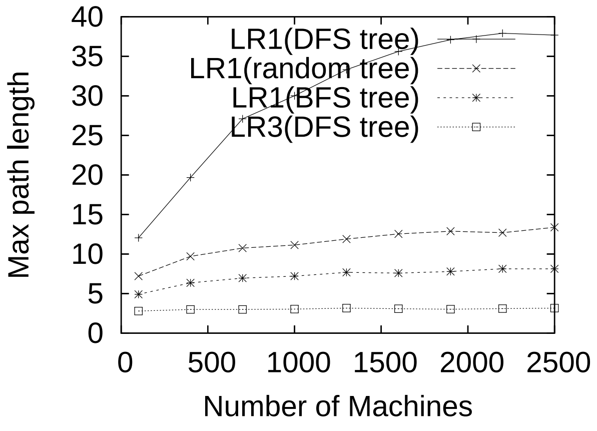
Table 3.6 shows the impact of selecting a logical ring with a shorter path length. For topology (4), LR1 results in a logical ring with a maximum path length of 4 hops, and LR2 results in a logical ring with a maximum path length of 2 hops. In addition to the results of LR1 and LR2, the table also includes results for topology (1), which is essentially a 1-hop ring. The results for topology (1) is provided for references since no logical ring algorithm can out-perform 1-hop ring. There are two observations from the table. First, the impact of path length on the performance is noticeable, but not very large in comparison to the impact of contention. Second, by minimizing the maximum path length of the ring on the Ethernet switched cluster, some performance improvement can be obtained. In general, the 2-hop ring performs better than the 4-hop ring, but worse than the 1-hop ring. Note that the theoretical lower bound time for all-to-all broadcast with a message size of $64KB$ on a

16-machine 100Mbps cluster is $\frac{15\times64\times1024\times8}{100\times10^6} = 78.6ms$. Considering the protocol overheads in MPI and TCP/IP layers as well as the software/hardware delays, the performance of LR2 (95.7ms) is very close to optimal.

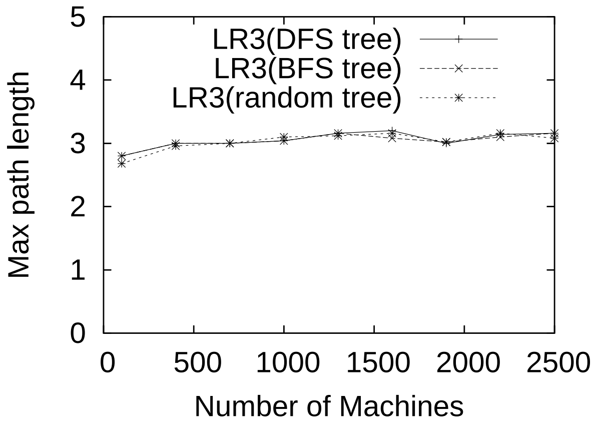
Next, a study of how these algorithms perform on large store-and-forward clusters is presented. This study is performed through simulation. In the simulation, it is assumed that the networks have arbitrary topologies. This allows us to evaluate the impacts of tree construction methods and to compare the results of different methods in computing logical rings. Since the logical rings found by all proposed algorithms are contention free, the maximum path length is used as the performance metric.

In the simulation, each random cluster is first generated. After the random cluster is generated, a tree construction method is used to build a spanning tree for the random cluster. Finally, the proposed algorithms are applied to compute logical rings. The random cluster is generated as follows: the number of machines and the number of switches for a cluster are decided. In the experiments, the ratio between the number of machines and the number of switches is fixed to be 5:1 (on average, each switch has five machines). The random connectivity among switches is generated using the *Georgia Tech Internetwork Topology Models (GT-ITM)* [86] with an average nodal degree of 4. Once the topology for the switches is generated, each machine is randomly distributed to any switch with equal probability. Three tree construction methods are considered: Breadth First Search (BFS) tree, Depth First Search (DFS) tree, and random tree. The BFS tree is created by first randomly selecting a root and then performing BFS on the graph. The DFS tree is created by first randomly selecting a root and then performing DFS. The random tree is created by repeatedly adding a random link to form a tree (if adding a link forms a loop, the link is not added).

Figure 3.25 shows the maximum path lengths in the logical rings computed using LR1 and LR3 with the three tree construction methods. Each point in the figure is the average of 50 random topologies. Figure 3.25 (a) shows that (1) the maximum path lengths of the logical rings computed by LR1 is much larger than those computed by LR3, and (2) the performance of LR1 depends heavily on the method used to construct the spanning tree. Using BFS trees yields much better results than using random trees, which in turn has much better results than DFS trees. This is because BFS trees usually have small tree heights while DFS trees are usually tall. Figure 3.25 (b) shows that LR3 produces much smaller maximum



(a)



(b)

Figure 3.25: Performance of LR1 and LR3 with different tree construction methods

path lengths in its rings. In all experiments performed on different kinds of random graphs, there is a very high probability that the maximum path length of each ring produced by LR3 is 3. This is reflected in Figure 3.25 (b): the average maximum path length is around 3, regardless of the cluster sizes. Furthermore, LR3 is not sensitive to the tree construction methods. BFS trees, DFS trees, and random trees yield very similar results. This indicates that LR3 is a robust algorithm for large networks.

3.4 Broadcast Operation

Broadcast is one of the most common collective communication operations. The broadcast operation requires a message from the *root* machine (the sender) to reach all other machines in the system at the end of the operation. The Message Passing Interface routine that realizes this operation is *MPI_Bcast* [51]. Broadcast algorithms are typically classified as either *atomic broadcast* algorithms or *pipelined broadcast* algorithms [2]. Atomic broadcast algorithms distribute the broadcast message as a whole through the network. Such algorithms apply to the cases when there is only one broadcast operation and the broadcast message cannot be split. When there are multiple broadcast operations or when the broadcast message can be split into a number of segments, a pipelined broadcast algorithm can be used, which distributes messages (segments) in a pipelined fashion.

This section investigates the use of pipelined broadcast to realize *MPI_Bcast* on Ethernet switched clusters when the message size is reasonably large. In this case, broadcasting a large message is carried out by a sequence of pipelined broadcasts with smaller message segments. In the pipelined broadcasts, communications on different branches of the logical broadcast tree can be active at the same time. To maximize the performance, communications that can potentially happen simultaneously should not share the same physical channel and cause network contention. Hence, the logical broadcast trees for pipelined broadcast should be contention-free. In this section, algorithms for computing various contention-free broadcast trees, which are suitable for pipelined broadcast on Ethernet switched clusters, are developed. The pipelined broadcast schemes are theoretically analyzed and empirically evaluated by comparing their performance with that of other commonly used broadcast algorithms. The results from the theoretical and experimental study, which is discussed later, indicate the following.

• When the message size is large, pipelined broadcast can be more effective than other broadcast schemes including the ones used in MPICH 2-1.0.1 (the latest MPICH release) [52] and LAM/MPI 7.1.1 (the latest LAM/MPI release) [44] to a large degree. Moreover, for large messages, the performance of pipelined broadcast on Ethernet switched clusters using a contention-free linear tree is close to the theoretical limit of the broadcast operation.

• Contention-free broadcast trees are essential for pipelined broadcast to achieve high performance on clusters with multiple switches. Pipelined broadcast using topology unaware broadcast trees may result in poor performance in such an environment.

• While it is difficult to determine the message segment size for pipelined broadcast to achieve the minimum communication time, finding one segment size that gives good performance is relatively easy since a wide range of message sizes can yield reasonably good performance for a given pipelined broadcast algorithm.

In the following, a number of commonly used broadcast algorithms are analyzed first. The algorithms for computing various contention-free broadcast trees on Ethernet switched clusters are then presented. Finally, the performance evaluation study is discussed.

3.4.1 Broadcast on Ethernet Switched Clusters

Let the number of machines in the broadcast operation be P. Let us assume that the time taken to send a message of size n between any two machines can be modeled as $T(n) =$

$\alpha + n \times \beta$, where α is the startup overhead and β is the per byte transmission time. When an $msize$-byte message is split into segments of sizes s_1, s_2, ..., and s_k, $T(s_1)+T(s_2)+...+T(s_k) \geq T(msize)$. Splitting a large message into small segments will increase the startup overheads and thus, the total communication time. Under the assumption that the startup overhead is insignificant in $T(s_i)$, $1 \leq i \leq k$, $T(s_1) + T(s_2) + ... + T(s_k) \approx T(msize)$.

Let the *communication completion time* be the duration between the time when the root starts sending and the time when the last machine receives the whole message. In the broadcast operation, each machine receives $msize$ data and the lower bound of the completion time is at least $T(msize)$. As will be shown later, this lower bound is approached by pipelined broadcast when $msize$ is sufficiently large.

Figure 3.26 shows some common broadcast trees, including linear tree, binary tree, k-ary tree, binomial tree, and flat tree. Common atomic broadcast algorithms include the flat tree and binomial tree algorithms. In the flat tree algorithm, the root sequentially sends the broadcast message to each of the receivers. The completion time is thus $(P-1) \times T(msize)$. In the binomial tree algorithm[28, 49], broadcast follows a hypercube communication pattern and the total number of messages that the root sends is $log(P)$. Hence, the completion time is $log(P) \times T(msize)$. In both of the flat tree and binomial tree algorithms, the root is busy throughout the communication and pipelined communication cannot be used to improve performance. Another interesting non-pipelined broadcast algorithm is the *scatter followed by all-gather* algorithm, which is used in MPICH [52]. In this algorithm, the $msize$-byte message is first distributed to the P machines by a scatter operation (each machine gets $\frac{msize}{P}$-byte data). After that, an all-gather operation is performed to combine messages to all nodes. In the scatter operation, $\frac{P-1}{P} \times msize$ data must be moved from the root to other nodes, and the time is at least $T(\frac{P-1}{P} \times msize)$. In the all-gather operation, each node must receive $\frac{P-1}{P} \times msize$-byte data from other nodes and the time is at least $T(\frac{P-1}{P} \times msize)$. Hence, the completion time for the whole algorithm is at least $2 \times T(\frac{P-1}{P} \times msize) \approx 2 \times T(msize)$.

Now, let us consider pipelined broadcast. Assume that the $msize$-byte broadcast message is split into X segments of size $\frac{msize}{X}$, broadcasting the $msize$-byte message is realized by X pipelined broadcasts of segments of size $\frac{msize}{X}$. To achieve good performance, the segment size, $\frac{msize}{X}$, should be small while keeping the startup overhead insignificant in $T(\frac{msize}{X})$. For example, in the experimental cluster, a segment size of $1KB$ results in good performance in most cases. Hence, when $msize$ is very large, X can be large.

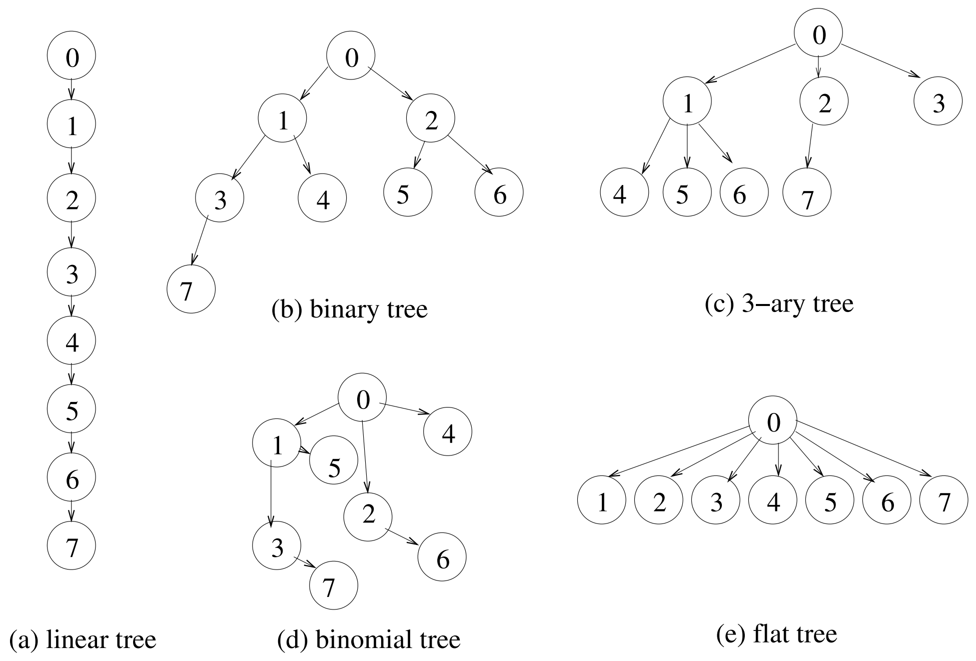


Figure 3.26: Examples of broadcast trees

The completion time for the X pipelined broadcasts depends on the broadcast tree, which decides the size of each pipeline stage and the number of pipelined stages. For simplicity, let us assume for now that there is no network contention in pipelined broadcast. Under the 1-port model, the size of a pipeline stage is equal to the time to send the number of messages that a machine must send in that stage, which is equal to the nodal degree of the machine in the broadcast tree. The number of pipelined stages is equal to the tree height. Let the broadcast tree height be H and the maximum nodal degree of the broadcast tree be D. The largest pipeline stage is $D \times T(\frac{msize}{X})$. The total time to complete the communication is roughly

$$(X + H - 1) \times (D \times T(\frac{msize}{X})).$$

When $msize$ is very large, X will be much larger than $H - 1$. In this case, $(X + H - 1)(D \times T(\frac{msize}{X})) \approx X \times (D \times T(\frac{msize}{X})) \approx D \times T(msize)$. This simple analysis shows that for large messages, trees with a small nodal degree should be used. For example, using a linear tree, shown in Figure 3.26 (a), $H = P$ and $D = 1$. The communication completion time is $(X + P - 1) \times T(\frac{msize}{X})$. When X is much larger than P, $(X + P - 1) \times T(\frac{msize}{X}) \approx T(msize)$, which is the theoretical limit of the broadcast operation.

Table 3.7: The performance of broadcast algorithms for very large messages

Algorithm	performance
Flat tree	$(P-1) \times T(msize)$
Binomial tree	$log_2(P) \times T(msize)$
scatter/allgather	$2 \times T(msize)$
Linear tree (pipelined)	$T(msize)$
Binary tree (pipelined)	$2 \times T(msize)$
k-ary tree (pipelined)	$k \times T(msize)$

Using the linear tree, the number of pipelined stages is P, which results in a long time to drain the pipeline when P is large. To reduce the number of pipelined stages, a general k-ary tree, that is, a tree with a maximum nodal degree of k, can be used. When $k = 2$, such trees are called binary trees. Assuming a complete binary tree is used, $H = log_2(P)$ and $D = 2$. The completion time is $(X+log_2(P)-1)\times 2\times T(\frac{msize}{X}) = (2X+2log_2(P)-2)\times T(\frac{msize}{X})$. When X is sufficiently large, $(2X + 2log_2(P) - 2) \times T(\frac{msize}{X}) \approx 2T(msize)$. When broadcasting a very large message, pipelined broadcast with a binary tree is not as efficient as that with a linear tree. However, when $2X + 2log_2(P) - 2 \leq X + P - 1$ or $X \leq P - 2log_2(P) + 1$, the binary tree is more efficient. In other words, when broadcasting a medium sized message, a binary tree may be more efficient than a linear tree. Under the 1-port model, when using general k-ary trees, $k > 2$, for pipelined broadcast, the size of the pipelined stage increases linearly with k while the tree height decreases proportionally to the reciprocal of the logarithm of k, assuming that trees are reasonably balanced such that the tree height is $O(log_k(P))$. Hence, under the 1-port model, it is unlikely that a k-ary tree, $k > 2$, can offer much better performance than a binary tree. For example, assuming a complete k-ary tree, $k > 2$, is used for pipelined broadcast, $H = log_k(P)$ and $D = k$. The completion time is $(X + log_k(P) - 1) \times (k \times T(\frac{msize}{X}))$, which is larger than the time for the complete binary tree for most practical values of X and P. The empirical study confirms this. Table 3.7 summarizes the performance of broadcast algorithms when the broadcast message size is very large.

3.4.2 Computing Contention-Free Broadcast Trees

As shown in Table 3.7, pipelined broadcast is likely to achieve high performance when the broadcast message is large. There are two obstacles that prevent this technique from being widely deployed. First, for pipelined broadcast to be effective, the logical broadcast tree must be contention-free. Finding a contention-free broadcast tree is a challenging task. Second, it is difficult to decide the optimal segment size for pipelined broadcast. In the following, algorithms for computing contention-free broadcast trees over physical tree topologies will be presented. Later, it will be shown that while deciding the optimal segment size may be difficult, it is relatively easy to find one size that achieves good performance.

Under the 1-port model, communications originated from the same machine cannot happen at the same time. Thus, a contention-free tree for pipelined broadcast only requires communications originated from different machines to be contention-free. Since each communication in a linear tree originates from a different machine, all communications in the contention-free linear tree must be contention-free. In a contention-free k-ary tree, communications from a machine to its (up to k) children may have contention.

3.4.2.1 Contention-Free Linear Trees

Let the machines in the system be n_0, n_1, ..., n_{P-1}. Let $F : \{0, 1, ..., P-1\} \rightarrow \{0, 1, ..., P-1\}$ be any one-to-one mapping function such that $n_{F(0)}$ is the root of the broadcast operation. $n_{F(0)}$, $n_{F(1)}$, ..., $n_{F(P-1)}$ is a permutation of n_0, n_1, ..., n_{P-1} and $n_{F(0)} \rightarrow n_{F(1)} \rightarrow n_{F(2)} \rightarrow$ ... $\rightarrow n_{F(P-1)}$ is a logical linear tree. The task is to find an F such that the communications in the logical linear tree do not have contention.

Let $G = (S \cup M, E)$ be a tree graph with S being the switches, M being the machines, and E being the edges. Let $P = |M|$ and n_r be the root machine of the broadcast. Let $G' = (S, E')$ be a subgraph of G that only contains switches and links between switches. A contention-free linear tree can be computed in the following two steps.

- Step 1: Starting from the switch that n_r is directly connected to, perform Depth First Search (DFS) on G'. Number the switches based on the DFS arrival order. An example numbering of the switches in the DFS order is shown in Figure 3.27. The switches are denoted as s_0, s_1, ..., $s_{|S|-1}$, where s_i is the ith switch arrived in the DFS traversal of G'. The switch that n_r attaches to is s_0.

- 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}. $n_r = n_{0,0}$. $X_i = 0$ when there is no machine attaching to s_i. The following logical linear tree is contention-free (this will be formally proved): $n_{0,0}(n_r) \rightarrow ... \rightarrow n_{0,X_0-1} \rightarrow n_{1,0} \rightarrow ... \rightarrow n_{1,X_1-1} \rightarrow ... \rightarrow n_{|S|-1,0} \rightarrow ... \rightarrow n_{|S|-1,X_{|S|-1}-1}$.

This algorithm is referred to as *Algorithm 4*. There exist many contention-free logical linear trees for a physical tree topology. It will be proved shortly that *Algorithm 4* computes one of the contention-free logical linear trees.

Lemma 9: Let $G' = (S, E')$ be the subgraph of G that contains only switches and links between switches. Let s_0, s_1, ..., $s_{|S|-1}$ be the DFS ordering of the switches, where s_i is the ith switch arrived in DFS traversal of G'. Communications in $\{s_0 \rightarrow s_1, s_1 \rightarrow s_2, ..., s_{|S|-2} \rightarrow s_{|S|-1}, s_{|S|-1} \rightarrow s_0\}$ are contention free. $\square$

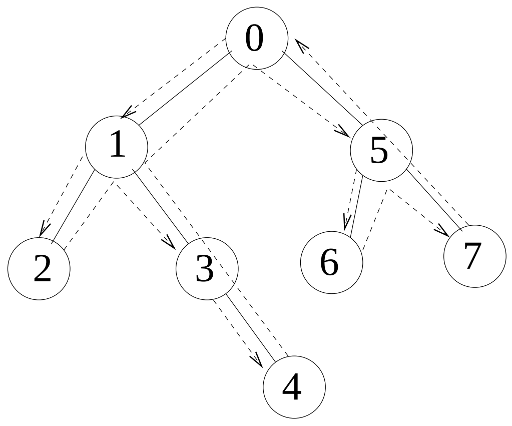


Figure 3.27: DFS numbering

The proof of Lemma 9 can be found in Section 3.3.2. Figure 3.27 shows an example. Clearly, communications in $\{s_0 \rightarrow s_1, s_1 \rightarrow s_2, s_2 \rightarrow s_3, s_3 \rightarrow s_4, s_4 \rightarrow s_5, s_5 \rightarrow s_0\}$ are contention-free.

Lemma 10: Let s_0, s_1, ..., $s_{|S|-1}$ be the DFS ordering of the switches. Let $0 \leq i < j \leq k < l \leq |S|-1$, $s_i \rightarrow s_j$ does not have contention with $s_k \rightarrow s_l$.

Proof: From Lemma 9, *path*$(s_i \rightarrow s_{i+1})$, *path*$(s_{i+1} \rightarrow s_{i+2})$, ..., *path*$(s_{j-1} \rightarrow s_j)$, *path*$(s_k \rightarrow s_{k+1})$, *path*$(s_{k+1} \rightarrow s_{k+2})$, ..., *path*$(s_{l-1} \rightarrow s_l)$ do not share any edge. It follows that *path*$(s_i \rightarrow s_{i+1}) \cup$ *path*$(s_{i+1} \rightarrow s_{i+2}) \cup ... \cup$ *path*$(s_{j-1} \rightarrow s_j)$ does not share any edge with *path*$(s_k \rightarrow s_{k+1}) \cup$ *path*$(s_{k+1} \rightarrow s_{k+2}) \cup ... \cup$ *path*$(s_{l-1} \rightarrow s_l)$. Since the graph is a tree, $path(s_i \rightarrow s_j) \subseteq path(s_i \rightarrow s_{i+1}) \cup path(s_{i+1} \rightarrow s_{i+2}) \cup ... \cup path(s_{j-1} \rightarrow s_j)$ and $path(s_k \rightarrow s_l) \subseteq path(s_k \rightarrow s_{k+1}) \cup path(s_{k+1} \rightarrow s_{k+2}) \cup ... \cup path(s_{l-1} \rightarrow s_l)$. Thus, $s_i \rightarrow s_j$ does not have contention with $s_k \rightarrow s_l$. □

Theorem 4: The logical linear tree obtained from *Algorithm 4* is contention free.

Proof: The linear tree is formed by grouping all machines attached to each switch together and ordering the switches based on the DFS order. Since each machine occurs in the linear tree exactly once, the link to and from each machine is used at most once in the linear tree. Thus, the intra-switch communications do not have contention. Since the switches are ordered based on DFS, from Lemma 10, the inter-switch communications do not have any contention. Hence, the linear tree is a contention-free linear tree. □

3.4.2.2 Contention-Free Binary Trees

As discussed earlier, since the tree height directly affects the time to complete the operation, the ideal binary tree for pipelined broadcast is one with the smallest tree height. Unfortunately, the problem of finding a contention-free binary tree with the smallest tree height is difficult to solve. In the following, a heuristic that computes contention-free binary trees while trying to minimize the tree heights is proposed. Although this heuristic may not find trees with the smallest tree heights, the simulation study indicates that the trees found by this heuristic are close to optimal. The heuristic is based on the contention-free linear tree obtained from *Algorithm 4*. The following lemma is the foundation of this heuristic.

Lemma 11: Let us re-number the logical linear tree obtained from *Algorithm 4* ($n_{0,0}(n_r) \rightarrow ... \rightarrow n_{0,X_0-1} \rightarrow n_{1,0} \rightarrow ... \rightarrow n_{1,X_1-1} \rightarrow ... \rightarrow n_{|S|-1,0} \rightarrow ... \rightarrow n_{|S|-1,X_{|S|-1}-1}$) as $m_0(n_r) \rightarrow m_1 \rightarrow ... \rightarrow m_{P-1}$. Let $0 \leq i < j \leq k < l \leq P-1$, communication $m_i \rightarrow m_j$ does not have contention with communication $m_k \rightarrow m_l$.

Proof: Let $m_i = n_{a,w}$, $m_j = n_{b,x}$, $m_k = n_{c,y}$, and $m_l = n_{d,z}$. Since $i < j \leq k < l$, $a \leq b \leq c \leq d$. $Path(m_i \rightarrow m_j)$ has three components: (m_i, s_a), $path(s_a \rightarrow s_b)$, and (s_b, m_j). $Path(m_k \rightarrow m_l)$ has three components: (m_k, s_c), $path(s_c \rightarrow s_d)$, and (s_d, m_l). When $a = b$, communication $m_i \rightarrow m_j$ does not have contention with communication $m_k \rightarrow m_l$ since (m_i, s_a) and (s_b, m_j) are not in $path(s_c \rightarrow s_d)$. Similarly, when $c = d$, communication $m_i \rightarrow m_j$ does not have contention with communication $m_k \rightarrow m_l$. When $a < b \leq c < d$, from Lemma 10, $path(s_a \rightarrow s_b)$ does not share edges with $path(s_c \rightarrow s_d)$. Hence, communication $m_i \rightarrow m_j$ does not have contention with communication $m_k \rightarrow m_l$ in all cases. □

Let $m_0 \rightarrow m_1 \rightarrow ... \rightarrow m_{P-1}$ be the linear tree obtained from *Algorithm 4*. For $0 \leq i \leq j \leq P-1$, let us denote sub-array $S(i,j) = \{m_i, m_{i+1}, ..., m_j\}$. The heuristic constructs contention-free binary trees for all sub-arrays $S(i,j)$, $0 \leq i \leq j \leq P-1$. Notice that for a sub-array $S(i,j)$, there always exists at least one contention-free binary tree since the linear tree is a special binary tree. Let $tree(i,j)$ represent the contention-free binary tree computed for $S(i,j)$. $Tree(0, P-1)$ is the binary tree that covers all machines. The heuristic builds $tree(i,j)$ with communications $m_a \rightarrow m_b$, $i \leq a < b \leq j$. Let $0 \leq i \leq j < k \leq l \leq P-1$, from Lemma 11, $tree(i,j)$ does not have contention with $tree(k,l)$.

Figure 3.28 shows the heuristic (*Algorithm 5*). In this algorithm, $tree[i][j]$ stores $tree(i,j)$, and $best[i][j]$ stores the height of $tree(i,j)$. Lines (2) to (12) are the base cases for binary trees with 1, 2, and 3 nodes. Note that under the 1-port model, $m_i \rightarrow m_{i+1}$ and $m_i \rightarrow m_{i+2}$ cannot happen at the same time. Hence, tree $\{m_i \rightarrow m_{i+1}, m_i \rightarrow m_{i+2}\}$ is the contention free binary tree for machines m_i, m_{i+1}, and m_{i+2}. Lines (13) to (26) iteratively compute trees that cover 4 to P machines. To compute $tree(i,j)$, $j > i+2$, the heuristic decides a k, $i+1 < k \leq j$, so that $tree(i,j)$ is formed by having m_i as the root, $tree(i+1, k-1)$ as the left child, and $tree(k,j)$ as the right child. Line (17) makes sure that $m_i \rightarrow m_k$ does not have contention with communications in $tree(i+1, k-1)$, which is crucial to ensure that the binary tree is contention-free. The heuristic chooses a k with the smallest $max(best[i+1][k-1], best[k][j]) + 1$ (lines (18) to (21)), which minimizes the tree height. At the end, $tree[0][P-1]$ stores the contention-free binary tree. Assume that the number of switches is less than P, the complexity of this algorithm is $O(P^4)$.

Theorem 5: The logical binary tree computed by *Algorithm 5* is contention-free.

Proof: It will be proved that, for all i and j, $0 \leq i \leq j \leq P-1$, (1) $tree[i][j]$ only consists of communications $m_a \rightarrow m_b$, $i \leq a < b \leq j$; and (2) $tree[i][j]$ is contention free.

Base case: It is trivial to show that trees with 1, 2, or 3 nodes satisfy the two conditions. For example, the 2-node tree rooted at node m_i contains nodes $\{m_i, m_{i+1}\}$ and one edge $m_i \rightarrow m_{i+1}$ (from lines (5) - (8) in Figure 3.28). This tree satisfies condition (1) since it only consists of communications $m_i \rightarrow m_{i+1}$. This tree is contention free since there is only one communication in the tree.

Induction case: Since $tree[i][j] = tree[i+1][k-1] \cup tree[k][j] \cup \{m_i \rightarrow m_{i+1}, m_i \rightarrow m_k\}$,

```
(1)  Let $m_0 \rightarrow m_1 \rightarrow ... \rightarrow m_{P-1}$ be the linear tree obtained from *Algorithm 4*.
(2)  for $(i = 0;\ i < P;\ i++)$ do
(3)      $best[i][i] = 0;\ tree[i][i] = \{\};$
(4)  enddo
(5)  for $(i = 0;\ i < P - 1;\ i++)$ do
(6)      $best[i][i+1] = 1;$
(7)      $tree[i][i+1] = \{m_i \rightarrow m_{i+1}\};$
(8)  enddo
(9)  for $(i = 0;\ i < P - 2;\ i++)$ do
(10)     $best[i][i+2] = 1;$
(11)     $tree[i][i+2] = \{m_i \rightarrow m_{i+1}, m_i \rightarrow m_{i+2}\};$
(12) enddo
(13) for $(j = 3;\ j < P;\ j++)$ do
(14)     for $(i = 0;\ i < P - j;\ i++)$ do
(15)         $best[i, i+j] = \infty;$
(16)         for $(k = i+2;\ k \leq i+j;\ k++)$ do
(17)             if ($m_i \rightarrow m_k$ does not have contention
                   with $tree[i+1][k-1]$) then
(18)               if $(best[i][i+j] > max(best[i+1][k-1],$
                                  $best[k][i+j]) + 1)$ then
(19)                 $best[i][i+j] = max(best[i+1][k-1],$
                                  $best[k][i+j]) + 1;$
(20)                 $index = k;$
(21)               endif
(22)             endif
(23)         enddo
(24)         $tree[i][i+j] = tree[i+1][index-1] \cup$
         $tree[index][i+j] \cup \{m_i \rightarrow m_{i+1}, m_i \rightarrow index\};$
(25)     enddo
(26) enddo
(27) $tree[0][P-1]$ stores the final result.
```

Figure 3.28: Heuristic to compute contention-free binary trees (Algorithm 5)

$i{+}2 < j$ and $i{+}1 < k \leq j$, $tree[i][j]$ only consists of communications $m_a \rightarrow m_b$, $i \leq a < b \leq j$.

From Lemma 11, communications in $tree[i+1][k-1]$ do not have contention with communications in $tree[k][j]$; $m_i \rightarrow m_{i+1}$ does not have contention with communications in $tree[k][j]$ and $tree[i+1][k-1]$; and $m_i \rightarrow m_k$ does not have contention with $tree[k][j]$. Thus, only $m_i \rightarrow m_k$ can potentially cause contention with communications in $tree[i+1][k-1]$. Since the algorithm makes sure that $m_i \rightarrow m_k$ does not cause contention with communications in $tree[i+1][k-1]$ (line (17)), there is no contention in $tree[i][j]$. $\Box$

Algorithm 5 can be easily extended to compute general k-ary trees. $S(i,j)$ can be basically partitioned into k sub-arrays which form the k subtrees. Precautions must be taken to prevent the communications from the root to a subtree from causing contention with communications in the subtrees.

The trees computed by *Algorithm 5* are evaluated through simulation. Figure 3.29 shows the results when applying *Algorithm 5* to clusters with different sizes (up to 1024 machines). Two cases are considered, on average 16 machines per switch and on average 8 machines per switch. For the 8 machines/switch case, a 1024-machine cluster has 128 switches. The cluster topologies are generated as follows. First, the size of the clusters to be studied is decided and the random tree topologies for the switches are generated by repeatedly adding random links between switches until a tree that connects all nodes is formed (links that violate the tree property are not added). After that, machines are randomly distributed to each switch with a uniform probability. For each size, 20 random topologies are generated and the average height of the 20 trees computed using *Algorithm 5* is reported. For comparison, the tree heights of complete binary trees for all sizes are also shown. As can be seen from the figure, the trees computed using *Algorithm 5* are not much taller than the complete binary tree, which indicates that the tree computed using *Algorithm 5* is close to optimal. Notice that the height of the complete binary tree is the lower bound of the height of the optimal contention-free binary tree. In most cases, contention-free complete binary trees do not exist.

3.4.3 Performance Evaluation

The performance of pipelined broadcast with different types of broadcast trees on different physical topologies are evaluated. The physical topologies used in the evaluation are shown in Figure 3.30. The topologies in Figure 3.30 are referred to as topologies (1), (2), (3), (4),

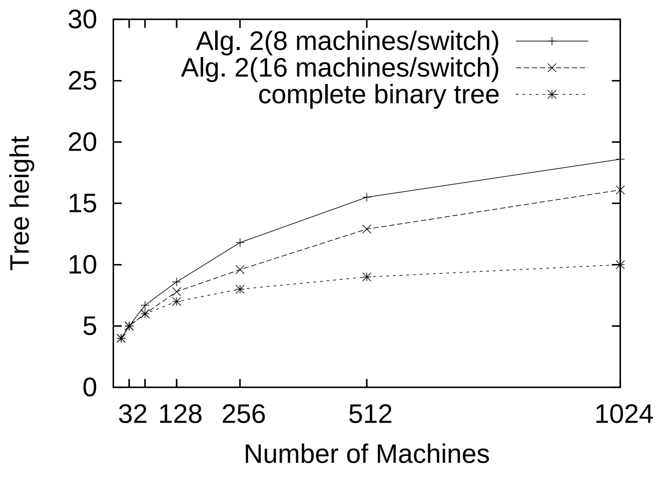


Figure 3.29: Performance of Algorithm 5

and (5). Topology (1) contains 16 machines connected by a single switch. Topologies (2), (3), (4), and (5) are 32-machine clusters with different network connectivity. Topologies (4) and (5) have exactly the same physical topology, but different node assignments.

The experiments are conducted on a 32-node Ethernet switched cluster. The nodes of the cluster are Dell Dimension 2400 with a 2.8GHz P4 processor, 128MB of memory, and 40GB of disk space. All machines run Linux (Fedora) with 2.6.5-1.358 kernel. The Ethernet card in each machine is Broadcom BCM 5705 with the driver from Broadcom. These machines are connected to Dell PowerConnect 2224 and Dell PowerConnect 2324 100Mbps Ethernet switches.

To evaluate the pipelined broadcast schemes, automatic routine generators are developed. These generators take the topology information as input and automatically generate customized *MPI_Bcast* routines that employ pipelined broadcast with different contention-free broadcast trees. The generated routines are written in C. They use MPICH point-to-point primitives and are compiled with the mpicc compiler in MPICH with no additional flags in the evaluation. The generated routines are compared with the original *MPI_Bcast* in LAM/MPI 7.1.1 [44] and MPICH 2-1.0.1 [52]. The code segment for the performance measurement is shown in Figure 3.31. Multiple iterations of *MPI_Bcast* are measured. Within each iteration, a barrier is added to prevent pipelined communication between iterations. Since only broadcasts with $msize \geq 8KB$ are considered, the barrier overhead is insignificant to

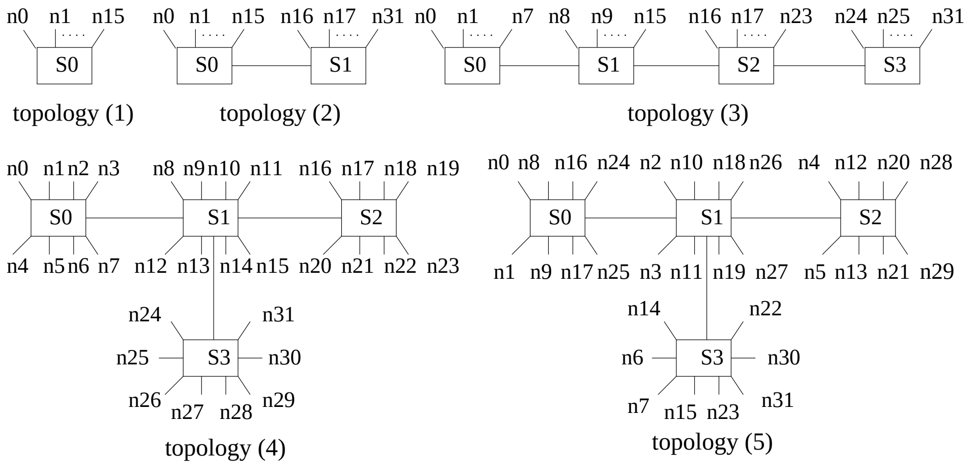


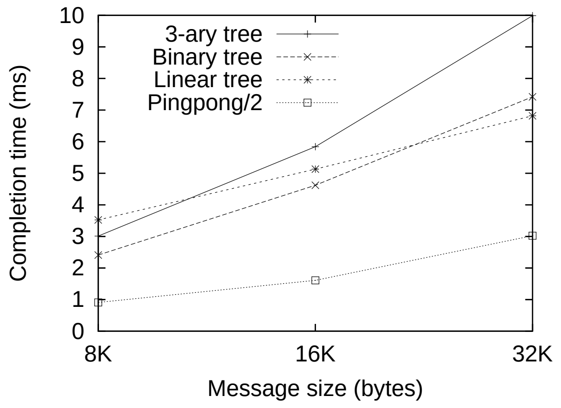
Figure 3.30: Topologies used in the evaluation

the total communication time. When reporting the performance of pipelined broadcast, by default, only the results with optimal segment sizes are reported, which are determined by an automatic tuning system (STAGE-MPI is discussed in the next chapter). Note that, as shown in Figure 3.36, while the optimal segment sizes are difficult to obtain, a wide range of segment sizes can yield performance close to the optimal.

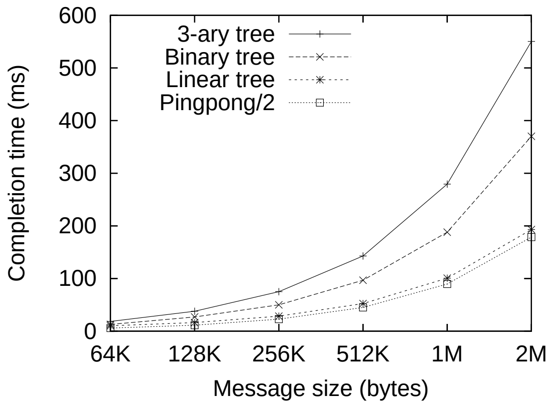
```
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();
for (count = 0; count < ITER_NUM; count ++) {
  MPI_Bcast(...);
  MPI_Barrier(...);
}
elapsed_time = MPI_Wtime() - start;
```

Figure 3.31: Code segment for measuring MPI_Bcast performance

Figure 3.32 shows the performance of pipelined broadcast using different contention-free trees on topology (1). The performance of pipelined broadcast on topologies (2), (3), (4), and (5) has a similar trend. As can be seen from the figure, when the message size is large ($\geq 32KB$), the linear tree offers the best performance. For medium sized messages ($8KB$ to $16KB$), the binary tree offers the best performance. In all experimental settings, the 3-ary



(a) Medium sized messages

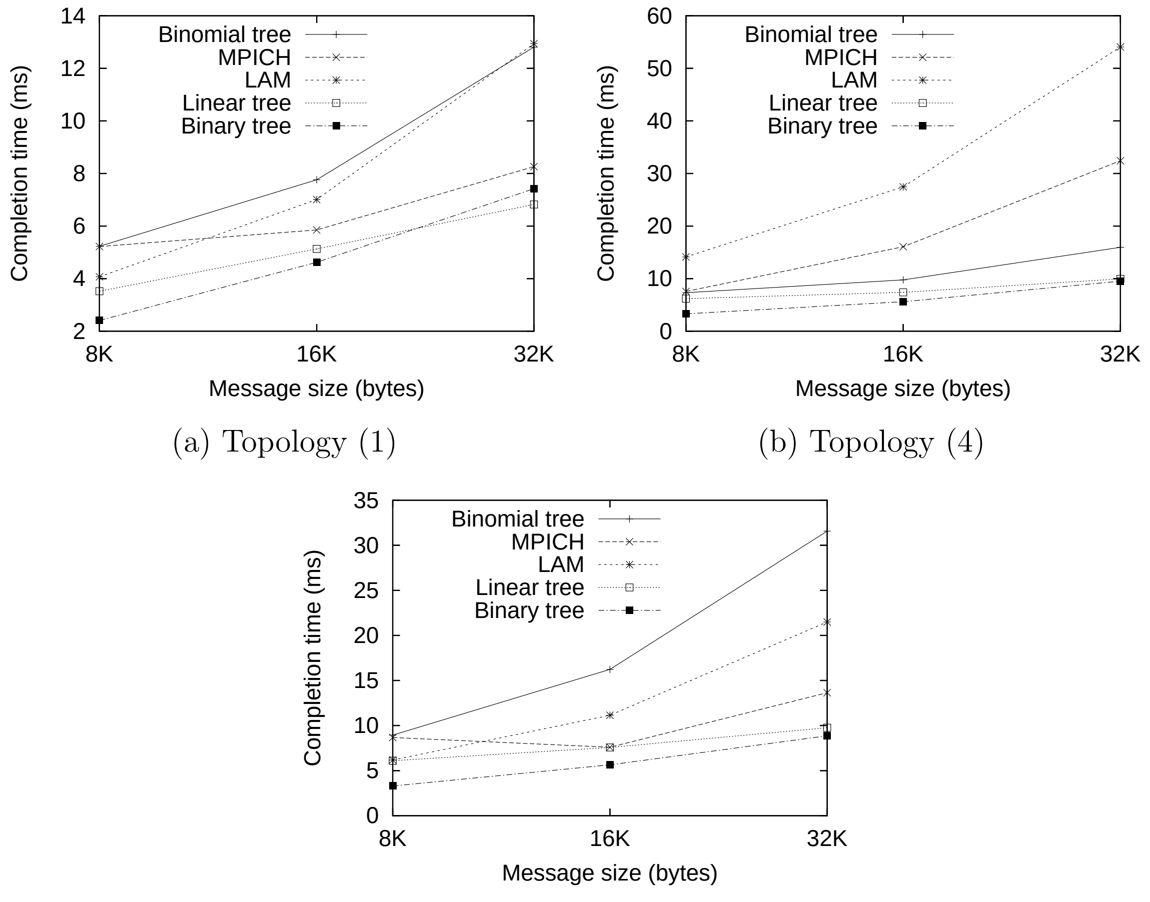


(b) Large sized messages

Figure 3.32: Performance of pipelined broadcast with different broadcast trees on topology (1)

tree is always worse than the binary tree, which confirms that k-ary trees, $k > 2$, are not effective. In the rest of the performance evaluation, only the performance of the linear tree and the binary tree is shown. The line titled "pingpong/2" in Figure 3.32 shows the time to send a single message of a given size between two machines, that is, $T(msize)$. When the message size is large ($\geq 256KB$), the communication completion time for linear trees is very close to $T(msize)$, which indicates that pipelined broadcast with the linear tree is clearly a good choice for Ethernet switched clusters when the message is large. The time for binary trees is about twice the time to send a single message.

Figures 3.33 and 3.34 compare the performance of pipelined broadcast using contention-free trees with the algorithms used in LAM/MPI and MPICH on topologies (1), (4), and (5). The results for topologies (2) and (3) are similar to those for topology (4). Since all algorithms run over MPICH except LAM, a binomial tree implementation (the algorithm used in LAM) over MPICH is also included in the comparison. MPICH uses the scatter followed by all-gather algorithm for large messages ($> 12KB$) and the binomial tree for small messages. When the message size is reasonably large ($\geq 8KB$), the pipelined broadcast routines significantly out-perform the none-pipelined broadcast algorithms used in LAM and MPICH. For topology (1) and (4), when the message size is large ($\geq 512KB$), MPICH has similar performance to the pipelined broadcast using binary trees. This is compatible with the analysis in Section 3.4.1 that both should have a completion time of around $2 \times T(msize)$. However, pipelined broadcast with linear trees is about twice as fast as MPICH when $msize \geq 512KB$. For topology (5), MPICH performs much worse than pipelined broadcast with binary trees. This is because the MPICH all-gather routine uses topology unaware algorithms and its performance is sensitive to the physical topology. Note that the MPICH all-gather routine changes algorithms when the broadcast message size is $512KB$. Hence, the performance curve for MPICH is non-continuous at this point ($512KB$) for some topologies.

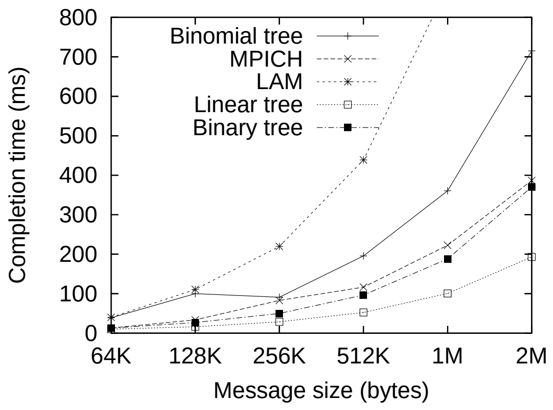


(a) Topology (1) (b) Topology (4)

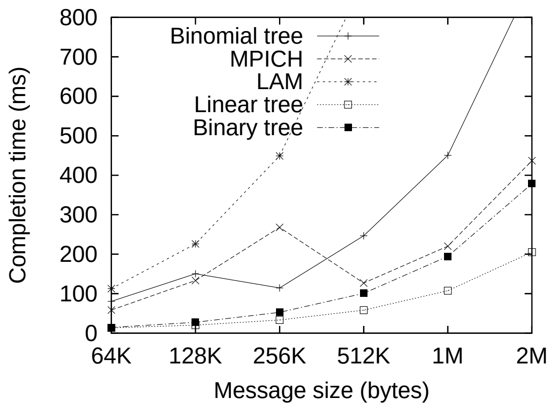
(c) Topology (5)

Figure 3.33: Performance of different broadcast algorithms (medium sized messages)

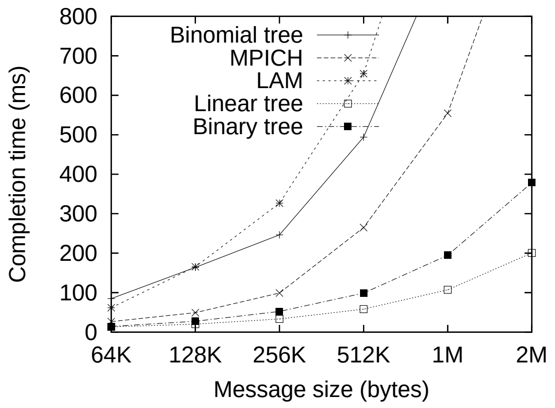
Figure 3.35 compares pipelined broadcast using contention-free trees with that using topology unaware trees. In the comparison, the topology unaware linear tree in [61, 78] is



(a) Topology (1)



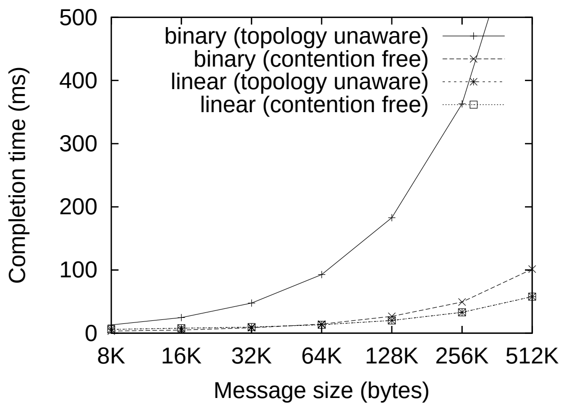
(b) Topology (4)



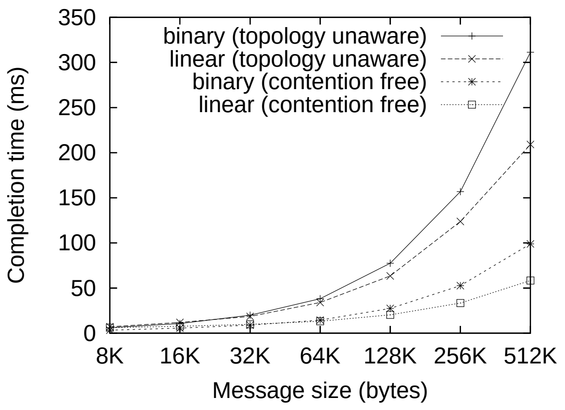
(c) Topology (5)

Figure 3.34: Performance of different broadcast algorithms (large sized messages)

used: $n_0 \rightarrow n_1 \rightarrow ... \rightarrow n_{P-1}$ (n_0 is the root). For topology unaware binary trees, the complete binary tree is assumed, where node n_k has children n_{2k+1} and n_{2k+2} and parent $n_{\frac{k-1}{2}}$. For topology (2), the topology unaware linear tree happens to be contention free. As a result, its performance is exactly the same as the contention-free linear tree. However, for topology (5), this is not the case, the topology unaware linear tree incurs significant network contention and its performance is much worse than the contention-free linear tree. For example, for broadcasting 1MB data on topology (5), the communication completion time is 58.4ms for the contention-free linear tree and 209.1ms for the topology unaware linear tree. The contention-free tree is 258% faster than the topology unaware tree. Topology unaware binary trees cause contention in all topologies except topology (1) in the experiments and

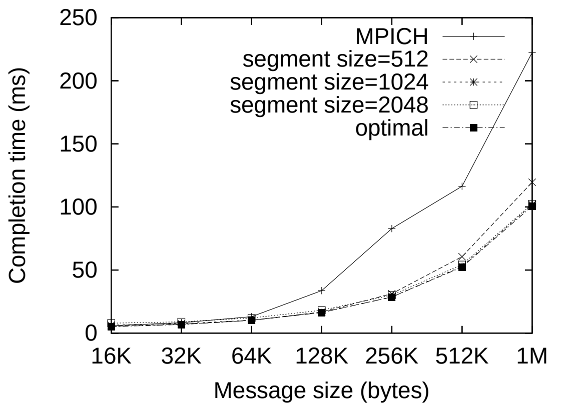


(a) Topology (2)

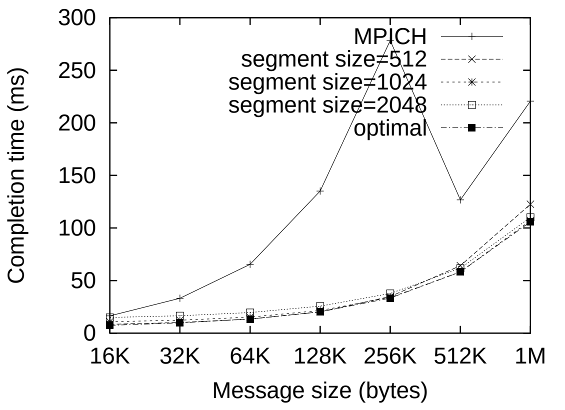


(b) Topology (5)

Figure 3.35: Contention-free broadcast trees versus topology unaware broadcast trees



(a) Topology (1)



(b) Topology (3)

Figure 3.36: Pipelined broadcast using linear trees with different segment sizes

their performance is significantly worse than the contention-free binary trees. These results indicate that to achieve high performance, contention-free broadcast trees must be used.

One of the important issues in pipelined broadcast is how to find a segment size that can achieve good performance. Figure 3.36 shows the impacts of segment sizes on the performance of pipelined broadcast with contention-free linear trees. The results for pipelined broadcast with binary trees have a similar trend. These figures indicate that pipelined broadcast is not very sensitive to the segment size. Changing from a segment size of $0.5KB$

to $2KB$ does not significantly affect the performance, especially in comparison to using a different algorithm. While finding the optimal segment size may be hard, deciding one size that can achieve good performance should not be very difficult since a wide range of segment sizes can result in near optimal performance.

3.5 Summary

In clusters of workstations where the network contention can significantly degrade the communication performance, of which Ethernet clusters are examples, developing efficient communication algorithms requires the consideration of the network topology. The performance evaluation studies of the generated routines for the all-to-all, all-gather, and broadcast operations show that the proposed topology-specific algorithms can in many cases outperform the ones included in the original LAM/MPI [44] and MPICH [52] libraries.

A natural question that may arise is whether or not such topology-specific communication algorithms are enough to achieve high performance in all cases. To discuss this issue, let us consider the following experiment. Using the same experimental setup and performance measurement scheme described in Section 3.2.4, the performance of three different all–to–all algorithms, including a topology-specific one, is considered on the same 32-node Ethernet cluster with a different topology configuration, which is shown in Figure 3.37. The topology-unaware algorithms used in the experiment are *pair* and *ring* while the topology-specific algorithm is *sender-all*. The pair algorithm partitions the all-to-all communication into $p-1$ steps (phases). In step i, node j sends/receives a message from/to node $i \oplus j$ (exclusive or). The ring algorithm is similar to the pair algorithm except that, in each step, node j sends a messages to node $(j+i)$ *mod* p and receives a message from node $(j-i)$ *mod* p. Details about the topology-specific algorithm can be found in Section 3.2.3.

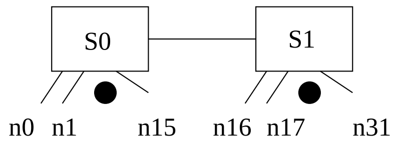


Figure 3.37: Example topology

The results of the experiment are shown in Table 3.8. For this particular topology, the

Table 3.8: Completion time of different all–to–all algorithms

msize	ring	pair	sender-all
2KB	44.83ms	56.60ms	116.7ms
4KB	93.00ms	103.1ms	160.4ms
8KB	184.0ms	198.2ms	266.0ms
16KB	366.1ms	395.0ms	446.3ms
32KB	731.0ms	783.1ms	830.0ms
64KB	1696ms	1881ms	1731ms
128KB	3383ms	3863ms	3626ms
256KB	5914ms	6254ms	6130ms
512KB	11.66s	15.60s	12.33s
1MB	23.41s	26.60s	25.09s

most efficient communication algorithm to realize the all–to–all operation across all message sizes is the ring algorithm. The table shows that the topology-specific algorithm (*Sender-all*) did not perform as well as the ring algorithm for the different message sizes, including large ones, although it can theoretically achieve the maximum bandwidth efficiency when the message size is sufficiently large. The results illustrate the following two important issues. First, the use of topology-specific communication algorithms can achieve high performance in many situations. However, there are situations (such as this particular topology) in which other algorithms, including topology-unaware ones, may actually offer higher performance. This indicates that developing platform-specific communication algorithms is important but not enough to *always* achieve the best performance. In any given situation, there are other factors besides the network topology that can significantly affect the performance of a communication algorithm. Second, the most efficient algorithm for one situation may be vastly different from that of another. To obtain an efficient collective routine, it is necessary to (1) have different communication algorithms that can potentially achieve high performance in different situations and (2) to be able to select the most efficient algorithm for a given situation.

CHAPTER 4

DELAYED FINALIZATION OF MPI COLLECTIVE COMMUNICATION ROUTINES

For collective communication routines to achieve high performance across platforms and applications, they must be able to adapt to platform and/or application configurations: the most efficient algorithm for a given configuration may be vastly different from that for a different configuration. The challenges in developing efficient MPI collective communication routines lie not so much in the development of an individual algorithm for a given situation. In fact, many efficient algorithms for various operations have been developed for different networks and different topologies. The major challenge lies in the mechanism for a library routine to adapt to platform and/or application configuration and to find the most efficient algorithm for a given configuration.

In this chapter, a new library implementation paradigm that allows MPI libraries to be more adaptive is proposed. This new approach is called the *delayed finalization of MPI collective communication routines* (DF). The idea of the DF approach is to decide the communication algorithms that will be used to carry out a collective operation after the platform and/or the application are known, which allows for applying architecture and/or application specific optimizations. A DF library includes two components: (1) an extensive set of communication algorithms; and (2) an automatic algorithm selection mechanism that performs the selection (or tuning) process on the algorithms to find the best communication algorithms for a given configuration. As mentioned previously, the implementation paradigm for a DF library is different from that for traditional MPI libraries. In particular, the DF library developers only implement the communication algorithms and the mechanisms to select the algorithms. They do not make decisions about which algorithms to use in an operation. The final communication algorithms for an operation are automatically selected by the algorithm selection mechanism. Figure 4.1 shows a high level view of a DF library.

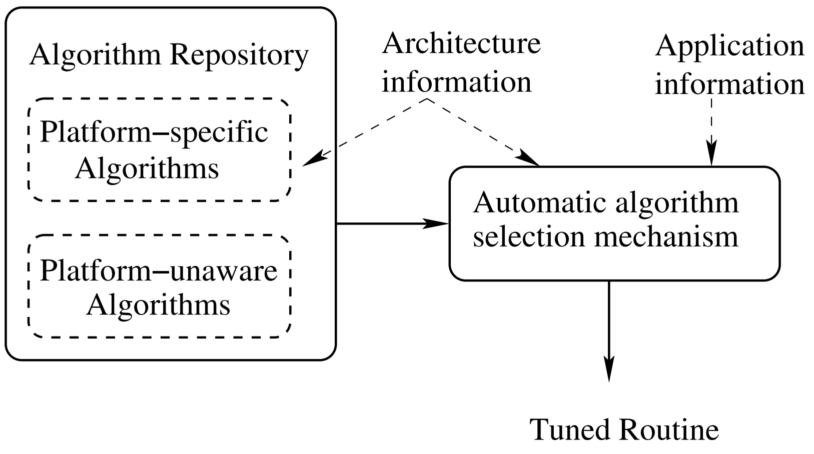

Algorithm Repository
Platform–specific Algorithms
Platform–unaware Algorithms
Architecture information
Application information
Automatic algorithm selection mechanism
Tuned Routine

Figure 4.1: High level view of a DF library

In contrast with traditional MPI libraries that only use platform-unaware communication algorithms, the algorithm repository of a DF library contains, for each operation, platform-specific communication algorithms in addition to an extensive set of platform-unaware communication algorithms. These communication algorithms are specifically optimized for the particular platform and can potentially achieve high performance in different situations.

Armed with the algorithm repository, a DF library uses an automatic algorithm selection mechanism to select the best algorithm for an operation in a given situation. Many algorithm selection schemes can be used. One example is to use a performance model to predict the performance of the communication algorithms in a particular situation and select the algorithm based on the prediction results. This model based approach can be very complicated since there are many factors that can affect the performance, and these factors can sometimes be very difficult to model. This thesis investigates the use of an empirical approach for the algorithm selection mechanism, where the performance of different communication algorithms is measured empirically and the best algorithm is selected based on the measured results. The employed empirical techniques in this thesis, which are variations of Automatic Empirical Optimization of Software (AEOS) [82], allows the aggregate effect of all factors that affect the performance of communication algorithms to be summarized in the measurements and used in the algorithm selection process.

To study the effectiveness of the DF approach and investigate the difficulties in implementing a DF library, two prototype DF libraries are developed. The first prototype DF library is the **S**tatic **T**uning and **A**utomatic **G**eneration of **E**fficient **MPI** collective routines (**STAGE-MPI**) system [20]. STAGE-MPI is a system consisting of an algorithm repository and a static empirical approach that performs the selection or tuning process at the library installation time. For each supported collective operation, STAGE-MPI examines the performance of different algorithms across different message sizes and decides the best algorithms, finds the message size range for each of the best algorithms, and automatically generates a final MPI routine that may include different algorithms for different message size ranges. The second prototype DF library consists of **S**elf **T**uned **A**daptive **R**outines for **MPI** collective operations (**STAR-MPI**) [22]. The routines in STAR-MPI are capable of carrying out the tuning process dynamically at run-time as the application executes. When a STAR-MPI routine is invoked in an application, an algorithm from STAR-MPI repository is chosen to realize the invocation and its performance is measured. After sufficient invocations of the routine and examining the performance of all algorithms, the best performing algorithm based on the measured results is used to realize the subsequent invocations.

This chapter is organized as follows. Section 4.1 discusses the STAGE-MPI [20] system. Section 4.2 discusses the application behavior and its impacts on collective operations [21] as a motivation for the STAR-MPI [22] library that is presented in Section 4.3. Finally, Section 4.4 summarizes the chapter.

4.1 STAGE-MPI

STAGE-MPI [20] is one realization of the DF approach. It integrates the two DF components: an algorithm repository and an automatic selection mechanism. The algorithm repository of STAGE-MPI includes topology-specific communication algorithms in addition to an extensive set of topology-unaware communication algorithms for each supported MPI collective operation. The topology-specific algorithms are automatically generated by routine generators included in STAGE-MPI. The automatic selection mechanism in STAGE-MPI is a variation of the Automated Empirical Optimization of Software (AEOS) technique [82], and it is used to select the best communication algorithm among the different algorithms in the repository. Using STAGE-MPI, the algorithm selection (tuning) process is performed at library installation time, where all algorithms for an operation are executed and their performance is measured, and the best performing algorithms are determined for different situations based on the measured results. The integration of the algorithm repository and the empirical selection mechanism enables STAGE-MPI to adapt to different architectures and construct efficient collective communication routines that are customized to the architectures.

Because the repository includes topology-specific algorithms designed for Ethernet switched clusters, STAGE-MPI achieves the best performance on Ethernet clusters. However, STAGE-MPI can also run on other platforms that support either LAM/MPI [44] or MPICH [52]. For these platforms, only the platform-unaware algorithms are included in its repository. The system currently tunes the following MPI collective communication routines: *MPI_Alltoall*, *MPI_Alltoallv*, *MPI_Allgather*, *MPI_Allgatherv*, *MPI_Allreduce*, *MPI_Bcast*, and *MPI_Reduce*. The routines produced by the system run on either LAM/MPI [44] or MPICH [52]. As will be shown later, evaluating the performance of the generated tuned routines shows that they are very robust and yield good performance for clusters with different network topologies. The tuned routines sometimes out-perform the routines in LAM/MPI and MPICH to a very large degree. In the following, the STAGE-MPI system is described first, and the performance evaluation of the system is presented second.

4.1.1 The System

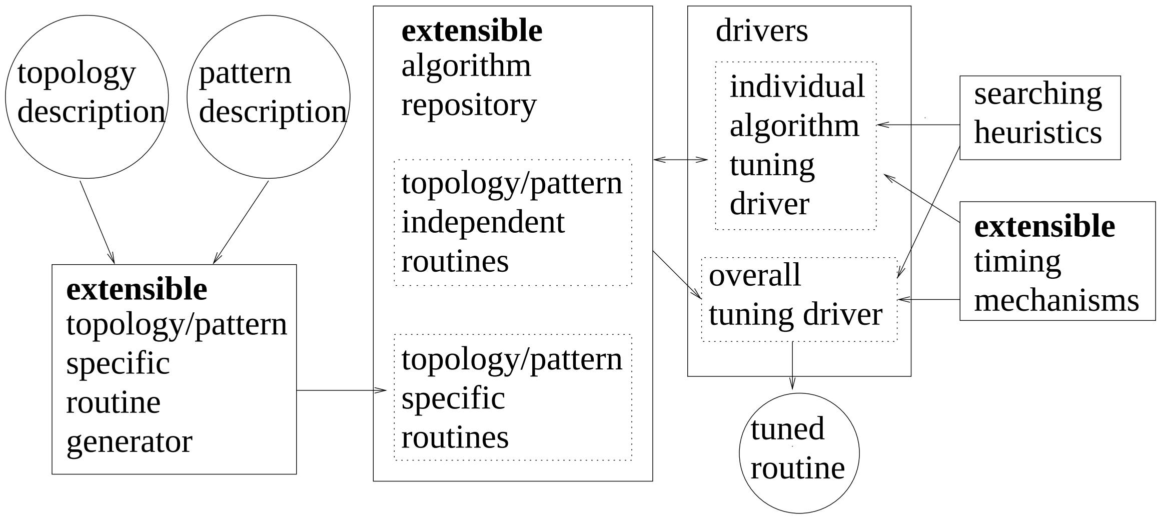

topology description
pattern description
extensible topology/pattern specific routine generator
extensible algorithm repository
topology/pattern independent routines
topology/pattern specific routines
drivers
individual algorithm tuning driver
overall tuning driver
searching heuristics
extensible timing mechanisms
tuned routine

Figure 4.2: System overview

As shown in Figure 4.2, there are five major components in STAGE-MPI: the extensible topology/pattern specific routine generator, the extensible algorithm repository, the search heuristics, the extensible timing mechanisms, and the drivers. The extensible topology/pattern specific routine generator takes topology description and sometimes pattern description and generates topology/pattern specific routines. For *MPI_Alltoall*, *MPI_Allgather*, *MPI_Allreduce*, *MPI_Reduce*, and *MPI_Bcast*, only the topology description is needed. The pattern description is also needed for *MPI_Alltoallv* and *MPI_Allgatherv*. This module is extensible in that users can provide their own routine generators with their own topology descriptors and pattern descriptors to replace STAGE-MPI built-in generators. For each supported MPI routine, the algorithm repository contains an extensive set of algorithms to be used in the tuning process. These algorithms include STAGE-MPI built-in topology/pattern unaware algorithms, topology/pattern specific algorithms generated by the routine generator module, and the results from the individual algorithm tuning drivers. Each routine in the repository may have zero, one, or more algorithm parameters. The repository is extensible in that it allows users to add their own implementations. The search heuristics determine the order in the search of the parameter space for deciding the best values for algorithm parameters. The extensible timing mechanisms determine how the timing results are measured. The timing results are used to guide the selection of the algorithms. This module is extensible in that users can supply their own timing mechanisms. The driver module contains individual algorithm tuning drivers and the overall tuning driver. The individual algorithms' tuning drivers tune algorithms with parameters, produce routines with no parameters (parameters are set to optimal values), and store the tuned routines back in the algorithm repository. The overall tuning driver considers all algorithms with no parameters and produces the final tuned routine. Next, each module will be described in more details.

4.1.1.1 Algorithm Repository

Before discussing the communication algorithms, the cost model that is used to give a rough estimate of the communication performance for the algorithms is described. It must be noted that some parameters in the cost model that can contribute significantly to the overall communication costs, such as sequentialization costs and network contention costs described below, are very difficult to quantify. In practice, they cannot be measured accurately since they are non-deterministic in nature. As a result, this cost model can be only used to justify the selection of algorithms in STAGE-MPI repository, but cannot be used to predict accurately which algorithm will be most effective for a given platform setting. STAGE-MPI uses an empirical approach to select the most effective implementations, and can even operate without the cost model.

Cost Model

A simple model to estimate the cost of collective communication algorithms has been widely used [77]. This simple model assumes that the time taken to send a message between any two nodes can be modeled as $\alpha + n\beta$, where α is the startup overhead (software overheads plus hardware delay), β is the per byte transmission time, and n is the number of bytes transferred. The simple model also assumes that communications between different pairs are unaware of each other, which is somewhat inaccurate for the all–to–all type of communications. When the message size is large, network contention may be the single factor that dominates the communication time. Barriers are often introduced in the communication operation to alleviate the network contention problem. To model the communications more accurately, these factors need to be taken into consideration. Let p be the number of processes. A slightly more complex model that reflects the following costs is used.

• *Per pair communication time.* The time taken to send a message of size n bytes between any two nodes can be modeled as $\alpha + n\beta$, where α is the startup overhead, and β is the per byte transmission time.

• *Sequentialization overhead.* Some algorithms partition the all-to-all type of communication into a number of phases. A communication in a phase can only start after the completion of some communications in the previous phases. This sequentialization overhead may limit the parallelism in the communication operation. The term θ denotes the sequentialization overhead between 2 phases. For a communication with m phases, the sequentialization overhead is $(m-1)\theta$.

• *Synchronization overhead.* There are two types of synchronizations in the algorithms: *light-weight* δ_l and *heavy-weight* δ_h. A light-weight barrier ensures that a communication happens before another while a heavy-weight barrier uses a system wide synchronization by calling *MPI_Barrier*. In most cases, δ_h is larger than δ_l, which is larger than θ.

• *Contention overhead.* Contention can happen in three cases: *node contention* γ_n when multiple nodes send to the same receiver, *link contention* γ_l when multiple communications use the same network links, and *switch contention* γ_s when the amount of data passing a switch is more than the switch capacity. The term $\gamma = \gamma_n + \gamma_l + \gamma_s$ denotes the sum of all contention costs.

Using this model, the time to complete a collective communication is expressed in the above five terms (α, β, θ, δ, and γ). The startup time and sequentialization overhead terms are important in algorithms for small messages while the bandwidth, synchronization costs, and contention overhead terms are important in algorithms for large messages.

In the rest of the following, p denotes the number of processes and n denotes the message size (passed as a parameter to routines *MPI_Alltoall*, *MPI_Allgather*, *MPI_Allreduce*, *MPI_Bcast*, and *MPI_Reduce*, that is, $n = sendcount * size\_of\_element$). Each node in the system can send and receive a message simultaneously, which is typical in Ethernet switched clusters. In the following, it is assumed by default that an algorithm does not have parameters, unless specified otherwise.

Algorithms for MPI_Alltoall

Simple algorithm. This algorithm basically posts all receives and all sends, starts the communications, and waits for all communications to finish. Let $i \rightarrow j$ denote the communication from node i to node j. The order of communications for node i is $i \rightarrow 0$, $i \rightarrow 1$, ..., $i \rightarrow p-1$. The estimated time for this algorithm is $(p-1)(\alpha + n\beta) + \gamma$.

Spreading Simple algorithm. This is similar to the simple algorithm except that the order of communications for node i is $i \rightarrow i+1$, $i \rightarrow i+2$, ..., $i \rightarrow (i+p-1)\ mod\ p$. This communication order may potentially reduce node contention. The estimated time is the same as that for simple algorithm except that the γ term might be smaller.

2D mesh algorithm. This algorithm organizes the nodes as a logical $x \times y$ mesh and tries to find the factoring such that x and y are close to $\sqrt{p}$. The all–to–all operation is carried out first in the x dimension and then in the y dimension. For all data to reach all nodes, the all–to–all operation is actually an all-gather operation that collects all data from each node to all nodes in each dimension. Thus, assuming $x = y = \sqrt{p}$, the message size for the all-gather operation in the x dimension is pn and the message size for the all-gather operation in the y dimension is $p\sqrt{p}n$. The estimated time for this algorithm is $(\sqrt{p}-1)(\alpha + pn\beta) + (\sqrt{p}-1)(\alpha + p\sqrt{p}n\beta) + \theta + \gamma = 2(\sqrt{p}-1)\alpha + (p-1)pn\beta + \theta + \gamma$.

Compared to the simple algorithms, the 2D mesh algorithm sends fewer messages, but more data. There is a θ term in the estimated time since communications are carried out in two phases.

3D mesh algorithm. This algorithm organizes the nodes as a logical $x \times y \times z$ mesh. Assume $x = y = z = \sqrt[3]{p}$. The estimated time is $3(\sqrt[3]{p} - 1)\alpha + (p-1)pn\beta + 2\theta + \gamma$. Compared to 2D mesh algorithm, this algorithm sends fewer messages, but consists of three phases, which introduce a 2θ sequentialization overhead.

Recursive doubling (rdb) algorithm. When the number of processes is a power of two, the recursive doubling algorithm is the extension of the 2D mesh and 3D mesh algorithms to the extreme: a $lg(p)$-dimensional mesh with 2 nodes in each dimension. This algorithm first performs an all-gather operation to collect all data from all nodes to each node. Each node then copies the right portion of the data to its receiving buffer. Details about recursive doubling can be found in [77]. When the number of nodes is a power of two, the estimated time is $lg(p)\alpha + (p-1)pn\beta + (lg(p) - 1)\theta + \gamma$. When the number of processes is not a power of two, the cost almost doubles [77]. Compared to the 3D mesh algorithm, this algorithm has a smaller startup time, but larger sequentialization overhead.

Bruck algorithm. This is another $lg(p)$-step algorithm that sends less extra data in comparison to the recursive doubling algorithm. Details can be found in [7, 77]. When the number of processes is a power of two, the estimated time is $lg(p)\alpha + \frac{np}{2}lg(p)\beta + (lg(p)-1)\theta + \gamma$. This algorithm also works with slightly larger overheads when the number of processes is not a power of two.

The above algorithms are designed for communication of small messages. Thus, the bandwidth and the contention terms in the estimated time are insignificant. To achieve good performance, the best trade-off must be found between the startup overhead and the sequentialization overhead. Next, algorithms designed for large messages are discussed.

Ring algorithm. This algorithm partitions the all-to-all communication into $p - 1$ steps (phases). In step i, node j sends a messages to node $(j + i)\ mod\ p$ and receives a message from node $(j - i)\ mod\ p$. Thus, this algorithm does not incur node contention if all phases are executed in a lock-step fashion. Since different nodes may finish a phase and start a new phase at different times, the ring algorithm only reduces the node contention (not eliminates it). The estimated time is $(p - 1)(\alpha + n\beta) + (p - 2)\theta + \gamma_n + \gamma_s + \gamma_l$.

Ring with light barrier algorithm. This algorithm adds light-weight barriers between the communications in different phases that can potentially cause node contention and eliminates such contention. The estimated time is $(p-1)(\alpha+n\beta)+(p-2)\delta_l+\gamma_s+\gamma_l$. Compared to the ring algorithm, this algorithm incurs overheads for the light-weight barriers while reducing the contention overheads.

Ring with MPI barrier algorithm. The previous algorithm allows phases to proceed in an asynchronous manner which may cause excessive data injected into the network. The ring with MPI barrier algorithm adds an MPI barrier between two phases and makes the phases execute in a lock-step fashion, resulting in a less likely switch contention. The estimated time is $(p-1)(\alpha+n\beta)+(p-2)\delta_h+\gamma_l$. Compared to the ring with light barrier algorithm, this algorithm incurs heavy-weight synchronization overheads while reducing the switch contention overheads.

Ring with N MPI barriers algorithm. Adding a barrier between every two phases may be an over-kill and may result in the network being under-utilized since most networks and processors can effectively handle a certain degree of contention. The ring with N MPI barriers algorithm adds a total of $1 \leq N\_mpi\_barrier \leq p-2$ barriers in the whole communication (a barrier is added every $\frac{p-1}{N\_mpi\_barrier+1}$ phases). This allows the contention overheads and the synchronization overheads to be compromised. The estimated time for this algorithm is $(p-1)(\alpha+n\beta)+N\delta_h+\gamma_n+\gamma_s+\gamma_l$. This algorithm has one parameter, the number of barriers ($N\_mpi\_barrier$). The potential value is in the range of 1 to $p-2$.

Pair algorithm. The algorithm only works when the number of processes is a power of two. This algorithm partitions the all-to-all communication into $p-1$ steps. In step i, node j sends and receives a message to and from node $j \oplus i$ (exclusive or). The estimated time is the same as that for the ring algorithm. However, in the pair algorithm, each node interacts with one other node in each phase compared to two in the ring algorithm. The reduction of the coordination among the nodes may improve the overall communication efficiency. Similar to the ring family algorithms, there are **pair with light barrier**, **pair with MPI barrier**, and **pair with N MPI barriers** algorithms.

The ring family and the pair family algorithms try to remove node contention and indirectly reduce other contention overheads by adding synchronizations to slow down communications. These algorithms are topology-unaware and may not be sufficient to eliminate link contention since communications in one phase may share the same link in the network. The topology-specific algorithm removes link contention by considering the network topology.

Topology-specific algorithm. A message scheduling algorithm that was developed in Section 3.2 is used. This algorithm finds the optimal message scheduling by partitioning the all–to–all communication into phases such that communications within each phase do not have contention, and a minimum number of phases are used to complete the communication. The estimated time for this algorithm depends on the topology.

Algorithms for MPI_Allgather

Since the *MPI_Allgather* communication pattern is a special all–to–all communication pattern, most of the all–to–all algorithms can be applied to perform an all-gather operation. STAGE-MPI includes the following all-gather algorithms that work exactly like their all-to-all counterparts (same estimated time), **simple**, **spreading simple**, **ring**, **ring with light barrier**, **ring with MPI barrier**, **ring with N MPI barriers**, **pair**, **pair with light barrier**, **pair with MPI barrier**, and **pair with N MPI barriers**. The following all-gather algorithms have different estimated times from their all–to–all counterparts: **2D mesh** with an estimated time of $2(\sqrt{p}-1)\alpha+(p-1)n\beta+\theta+\gamma$, **3D mesh** with an estimated time of $3(\sqrt[3]{p}-1)\alpha+(p-1)n\beta+2\theta+\gamma$, and **Recursive doubling (rdb)**. When the number of processes is a power of two, the estimated time of rdb is $lg(p)\alpha+(p-1)n\beta+(lg(p)-1)\theta+\gamma$. The repository also includes the following algorithms:

Bruck algorithm. The bruck all-gather algorithm is different from the bruck all-to-all algorithm. Details can be found in [7, 77]. When the number of processes is a power of two, the estimated time is similar to the recursive doubling algorithm. The time is better than that of recursive doubling when the number of processes is not a power of two.

Gather-Bcast algorithm. This algorithm first gathers all data to one node and then broadcasts the data to all nodes. Assume that the gather and broadcast operations use the binary tree algorithm, the estimated time is $lg(p)(\alpha+n\beta)+(lg(p)-1)\theta+\gamma$ for gather and $lg(p)(\alpha+pn\beta)+(lg(p)-1)\theta+\gamma$ for broadcast.

Topology-specific algorithm. The topology-specific logical ring (TSLR) all–gather algorithm that was developed in Section 3.3 is used. The algorithm constructs a contention-free logical ring pattern. To complete an all–gather operation, the algorithm repeats the logical ring communication pattern $p-1$ times. In the first iteration, each node sends its own data to the next adjacent node in the logical ring. In the following iterations, each node forwards what it received in the previous iteration to its adjacent node. Details about this algorithm can be found in Section 3.3. The estimated time is $(p-1)(\alpha+n\beta)+(p-2)\theta+\gamma_s$. Note that MPICH [52] uses a topology- unaware logical ring (LR) algorithm that operates in the same way as the TSLR algorithm. However, without considering the network topology, the ring pattern in the MPICH algorithm may result in severe network contention, which degrades the performance.

Algorithms for MPI_Allreduce

The implementations for *MPI_Allreduce* assume the reduction operation is commutative.

Reduce-Bcast algorithm. The algorithm first performs a reduction to a node and then broadcasts the results to all nodes. The completion time depends on the reduce and broadcast routines.

All-gather based algorithm. The algorithm first gathers all data to all nodes. Then, each node performs the reduction locally. This algorithm uses the tuned *MPI_Allgather* routine, which can be topology-specific. The time for this algorithm depends on the tuned all–gather routine.

Recursive doubling (rdb) algorithm. This algorithm is similar to the all-gather based algorithm except that the reduction operation is performed while the data are being distributed. Since the computation costs are ignored, the estimated time is the same as recursive doubling for all-gather.

MPICH Rabenseifner (MPICH Rab) algorithm. This algorithm completes in two phases: a reduce-scatter followed by an all-gather. The reduce-scatter is realized by recursive halving, which has a similar estimated time as recursive doubling. The all-gather is realized by recursive doubling. The time for this algorithm is roughly 2 times that of rdb algorithm for all-gather with a message size of $\frac{n}{p}$. More details about the algorithm can be found in [64].

Rabenseifner variation 1 (Rab1) algorithm. This is a Rabenseifner algorithm with the all-gather operation performed using the tuned all-gather routine. This algorithm may be topology-specific since the tuned all-gather routine may be topology specific.

Rabenseifner variation 2 (Rab2) algorithm. In this variation, the reduce-scatter operation is realized by the tuned all–to–all routine with a message size of $\frac{n}{p}$ and the all–gather operation is realized by the tuned all–gather routine with a message size of $\frac{n}{p}$.

Algorithms for MPI_Bcast

Linear tree algorithm. Let p_0 be the root of the broadcast. The algorithm follows a store-and-forward communication pattern: $p_0 \rightarrow p_1 \rightarrow ... \rightarrow p_{p-1}$. The broadcast message is sent from p_0 to p_1 first, p_1 to p_2 second, and so forth until the message reaches the last processor p_{p-1}. The estimated time for this algorithm is $(p-1)(\alpha + n\beta)$.
Flat tree algorithm. In this algorithm, the root sequentially sends the broadcast message to each of the receivers. Thus, the estimated time is $(p-1)(\alpha + n\beta)$.
Binomial tree algorithm. The broadcast operation in this algorithm follows a hypercube communication pattern and the total number of messages the root sends is $lg(p)$. Hence, the completion time is $lg(p)(\alpha + n\beta) + \gamma_l + \gamma_s$. Details about this algorithm can be found in [28, 49].
Scatter-Allgather algorithm. Using this algorithm, a broadcast operation is realized by a scatter operation followed by an all–gather operation. First, the message is distributed to the p processors, where each processor gets $\frac{n}{p}$ message. After that, an all–gather operation is performed to combine the $\frac{n}{p}$ messages to all nodes. The total completion time for this algorithm depends on the all–gather routine.
Topology-specific algorithms. There are two topology-specific algorithms: pipelined linear tree and pipelined binary tree. These algorithms are intended for broadcasts of large messages. Let H be the height of the logical broadcast tree and D be the nodal degree. In pipelined broadcast, the broadcast message can be broken into X messages each of size $\frac{n}{X}$. The total time to complete the pipelined communication is roughly $(X + H - 1)\times$ $(D\times$ $(\alpha + \frac{n}{X}\beta))$. When the message size is very large, the startup cost α can be ignored, and X can be much larger than $H-1$. Thus, the completion time is roughly $X \times D \times \frac{n}{X}\beta \approx D \times n\beta$. Thus, when using a logical linear tree ($D = 1$), the broadcast completion time is roughly $n\beta$, and in the case of a binary tree ($D = 2$), the completion time is $2n\beta$. Detailed discussion of the two algorithms can be found in Section 3.4.

Algorithms for MPI_Reduce

The communication in the reduction operation is the reverse of that in the broadcast operation: nodes send messages to root. Hence, all broadcast algorithms can be used to perform the reduce operation with the communication in the reversed direction. Note that since messages are sent from multiple nodes to the root at the same time, node contention can be imminent. In addition, depending on the communication algorithm, link and switch contention can occur.

Linear tree algorithm. Let p_0 be the root of the reduce operation. The algorithm operates as follows: p_{p-1} sends its data to p_{p-2} who will perform the reduction operation on its own data and the received data; p_{p-2} sends the new reduced data to p_{p-3}, which in turn applies the reduction on its own data and received data. The process continues until p_0 receives the so far reduced data and performs the reduction operation on it. The estimated time for this algorithm is $(p-1)(\alpha + n\beta)$.

Flat tree algorithm. This algorithm is similar to the version used in the broadcast operation. The difference is that the root sequentially receives the message to be reduced from each of the senders. The estimated time is $(p-1)(\alpha + n\beta) + \gamma$. Note that, unlike the flat tree broadcast algorithm, contention $(\gamma_n + \gamma_l + \gamma_s)$ may occur in this algorithm as nodes send their data to the root.

Binomial tree algorithm. This is also similar to the version used in the broadcast operation except that the data flows in the opposite direction: from nodes to root. The completion time is $lg(p)(\alpha + n\beta) + \gamma$.

MPICH Rabenseifner (MPICH Rab) algorithm. This algorithm is similar to the one used for the all-reduce operation. The algorithm completes in two phases: a reduce-scatter followed by a gather. The reduce-scatter is realized by recursive halving while the gather is realized by binomial tree. More details of the algorithm can be found in [64].

Topology-specific algorithms. Similar to the pipelined algorithms for the broadcast operation, there are pipelined linear tree and pipelined binary tree algorithms for the reduce operation. Note that the reduction operation is applied at intermediate nodes as the pipelined data segments flow towards the root. The estimated time for the algorithms is the same as the ones used in the broadcast operation, with the addition of the computation cost of performing the reduction operation.

Algorithms for MPI_Alltoallv

Most of the topology-unaware all–to–all algorithms can be used to realize the *MPI_Alltoallv* operation. STAGE-MPI contains the all–to–allv version of the following all–to–all algorithms: **simple**, **spreading simple**, and the **ring** and **pair** families algorithms.

Topology-specific algorithms. There are two topology-specific *MPI_Alltoallv* algorithms:

greedy algorithm and **all–to–all based** algorithm. These two algorithms are extensions of the algorithms with the same names in the CCMPI package developed in [38]. Since *MPI_Alltoallv* supports many-to-many communication with different message sizes, there are three issues in realizing this communication: balancing the load, reducing network contention, and minimizing the number of phases. The greedy algorithm focuses on balancing the load and reducing network contention while the all–to–all based algorithm considers all three issues when scheduling messages. Details about these algorithms can be found in [38].

Algorithms for MPI_Allgatherv

Most of the topology-unaware all-gather algorithms are extended to the all-gatherv operation. The algorithms include the **simple**, **recursive doubling**, **ring** and **pair** families algorithms. The topology-specific algorithm is based on the topology-specific all-gather algorithm.

4.1.1.2 Timing Mechanisms

The timing mechanisms constitute the most critical component in STAGE-MPI as it decides how the performance of a routine is measured. Since the measurement results are used to select the best algorithms, it is essential that the timing mechanism gives accurate timing results. Unfortunately, the performance of a communication routine depends largely on the application behavior. STAGE-MPI makes the timing module extensible, which allows users to supply application specific timing mechanisms that can closely reflect the application behavior. In addition, STAGE-MPI has three built-in timing mechanisms from which users can select the one that best matches their needs. These built-in timing mechanisms are variations of the Mpptest approach [32], and the timing results of these mechanisms are fairly consistent and repeatable for the routines STAGE-MPI currently supports.

Figure 4.3 shows a code segment for each of the different built-in timing mechanisms supported in STAGE-MPI. Part (a) of the figure represents the plain *Mpptest* scheme, where the performance of multiple invocations of a routine is measured. Part (b) shows the *Mpptest + barrier* scheme. In this scheme, a barrier is added within each iteration to prevent pipelined communication between the iterations. Finally, Part (c) describes the *Mpptest + computation* performance measurement scheme. To resemble the interaction between computations and communications in real applications, in this scheme, a computation code segment is added

```
MPI_Barrier(MPI_COMM_WORLD)
start = MPI_Wtime();
for (i = 0;  i < ITER;  i++)
    MPI_Alltoall(..);
elapsed = MPI_Wtime() − start;
```

(a) Mpptest

```
elapsed = 0;
for (i = 0;  i < ITER;  i++){
    MPI_Barrier(MPI_COMM_WORLD)
    start = MPI_Wtime();
    MPI_Alltoall(..);
    elapsed += (MPI_Wtime() − start);
}
```

(b) Mpptest + barrier

```
elapsed = 0;
for (i = 0;  i < ITER;  i++){
    /* computation */
    for (j = 0;  j < BOUND;  j++)
       for (k = 0; k  < 1000; k++)
           a[k] = b[k+1] + a[k−1];

   start = MPI_Wtime();
   MPI_Alltoall(..);
   elapsed += (MPI_Wtime() − start);
}
```

(c) Mpptest + computation

Figure 4.3: Three different built-in timing mechanisms in STAGE-MPI

within each iteration. As shown in the figure, the execution time for the computation can be controlled by setting the variable $BOUND$.

4.1.1.3 Search Heuristics

The search heuristics decide the order that the parameter space is searched to find the best algorithm parameters, which decide the time to tune a routine. In STAGE-MPI, the ring with N MPI barriers and pair with N MPI barriers algorithms have one algorithm parameter, $N\_mpi\_barrier$, which has a small solution space. Similarly, pipelined linear and binary tree algorithms for the broadcast and reduce operations have one algorithm parameter, *segment_size*, of small solution space. STAGE-MPI only supports a linear search algorithm, that is, deciding the best solution for each parameter by linearly trying out all potential values. STAGE-MPI handles multiple parameters cases by assuming that the parameters are independent from each other. The linear search algorithm is sufficient for STAGE-MPI.

4.1.1.4 Drivers

The process for tuning *MPI_Alltoall*, *MPI_Allgather*, *MPI_Allreduce*, *MPI_Reduce*, and *MPI_Bcast* is different from that for tuning *MPI_Alltoallv* and *MPI_Allgatherv*, and that is depicted in Figure 4.4. The process contains four steps. In the first step, STAGE-MPI prompts the user for inputs, which include (1) the routine(s) to tune, (2) whether to consider topology-specific routines, (3) which routine generator to use (users can choose the built-in generator or supply their own generator), (4) the topology description file, (5) the timing mechanism (users can choose among the built-in ones or supply their own timing program). In the second step, STAGE-MPI generates the topology-specific routines if requested. In the third step, algorithms with parameters are tuned. The tuning is carried out as follows. First, for a set of fixed message sizes (currently set to $1B$, $64B$, $512B$, $1KB$, $2KB$, $4KB$, $8KB$, $16KB$, $32KB$, $64KB$, $128KB$, and $256KB$), the linear search algorithm is used to find the best performing value for each parameter for each of the sizes. STAGE-MPI then examines each pair of adjacent message sizes. If the best parameter values are the same for the two sizes, STAGE-MPI will use the parameter values for all message sizes in the range between the two sizes. If different best parameter values are used for the two points of the message sizes, a binary search algorithm is used to decide the crossing point where the parameter value should be changed. For each operation, STAGE-MPI assumes the same algorithm when the message size is larger than or equal to $256KB$. This step generates a tuned routine for the particular algorithm with the best parameter values set for different ranges of message sizes. This tuned routine is stored back in the algorithm repository as an algorithm without parameters. In the last step, all algorithms with no parameters are considered. The process is similar to that in step 3. The only difference is that instead of tuning an algorithm with different parameter values, this step considers different algorithms. Figure 4.5 (a) shows an example of the final generated all-to-all routine.

When tuning *MPI_Alltoallv* and *MPI_Allgatherv*, STAGE-MPI also asks for the pattern description file in addition to other information. The routine generator uses both the topology and pattern information and produces a routine for the specific topology and pattern. Tuning algorithms with parameters in the third step is straight-forward, STAGE-MPI just measures the performance of all potential values for a parameter for the specific pattern and decides the best parameter values. Finally, the last step considers all algorithms and selects the best algorithm. STAGE-MPI potentially generates a different implementation for each invocation of a routine. To produce a compact routine for an application, STAGE-MPI allows the pattern description file to contain multiple patterns, which may correspond to the sequence of invocations of the routine in the application. This pattern file can be

```
Step 1: Prompt the user for the following information:
 1.1 which routine to tune;
 1.2 whether or not to include topology-specific routines;
 1.3 which routine generator to use;
 1.4 the topology description file;
 1.5 which timing mechanism to use;

Step 2: Generate the topology-specific routines.

Step 3: Tune algorithms with parameters.
 3.1 Decide the best parameter values for a set of message sizes
     (currently 1B, 64B, 256B, 1KB, 2KB, 4KB, 8KB,
     16KB, 32KB, 64KB, 128KB, 256KB).
 3.2 Find the exact message sizes when the best
     parameter values are changed (binary search).
 3.3 Generate one routine with the best parameters set
     and store it in the algorithm repository.

Step 4: Final tuning, generate the final routine.
 /* only considers algorithms with no parameters */
 4.1 Decide the best algorithm for a set of message sizes
     (currently 1B, 64B, 256B, 1KB, 2KB, 4KB, 8KB,
     16KB, 32KB, 64KB, 128KB, 256KB).
 4.2 Find the exact message sizes when the best
     algorithms are changed using (binary search).
 4.3 Generate the final routine with the best algorithms
     selected for different message ranges.
```

Figure 4.4: A tuning process example

created by profiling the program execution. STAGE-MPI then creates a sequence of tuned implementations for the sequence of patterns. To reduce the code size, before a tuned routine is generated for a pattern, the pattern is compared with other patterns whose routines have been generated. If the difference is under a threshold value, the old tuned routine will be used for the new pattern. Figure 4.5 (b) shows an example of the final generated all–to–allv routine for an application.

```
int alltoall_tuned(...) {
  if ((msg_size >= 1) && (msg_size < 8718))
     alltoall_simple(...);
   else if ((msg_size >= 8718) && (msg_size < 31718))
     alltoall_pair_light_barrier(...);
   else if ((msg_size >= 31718) && (msg_size < 72032))
     alltoall_pair_N_mpi_barrier_tuned(...);
   else if (msg_size >= 72032)
     alltoall_pair_mpi_barrier(sbuff, scount, ...);
}
```

(a) An example tuned MPI_Alltoall routine

```
int alltoallv_tuned(...) {
   static int pattern = 0;
   if (pattern == 0) {
     alltoallv_tspecific_alltoall(...); pattern++;
   } else if ((pattern >= 1) && (pattern < 100)) {
     alltoallv_ring(...); pattern ++;
   } else { MPI_alltoallv(...); pattern++; }
}
```

(b) An example tuned MPI_Alltoallv routine

Figure 4.5: Examples of tuned routines

4.1.2 Performance Evaluation

The experiments are performed on a 32-node Ethernet-switched cluster. The nodes of the cluster are Dell Dimension 2400 with a 2.8GHz P4 processor, 128MB of memory, and 40GB of disk space. All machines run Linux (Fedora) with 2.6.5-1.358 kernel. The Ethernet card in each machine is Broadcom BCM 5705 with the driver from Broadcom. These machines are connected to Dell PowerConnect 2224 and Dell PowerConnect 2324 100Mbps Ethernet switches.

Experiments on many topologies are conducted. In all experiments, the tuned routines are robust and offer high performance. Three representative topologies are used to report

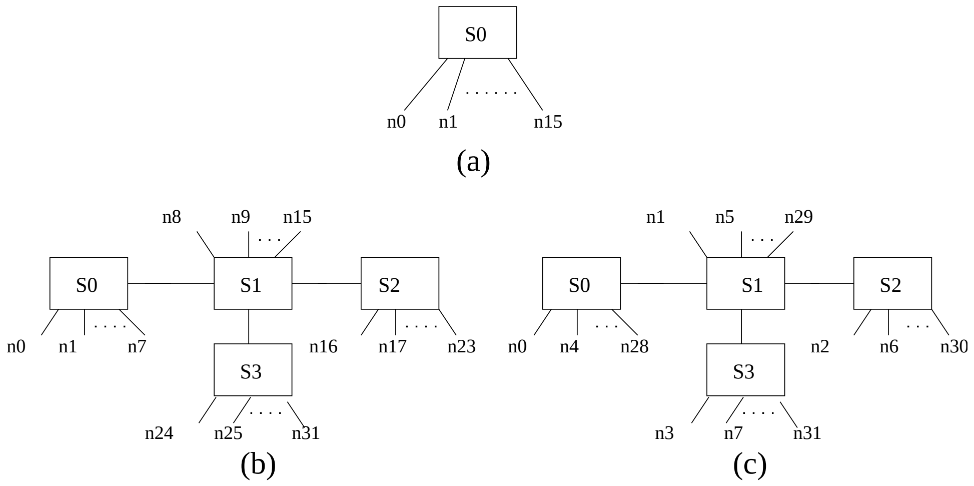


Figure 4.6: Topologies used in the experiments

the results, which are shown in Figure 4.6. Figure 4.6 (a) is a 16-node cluster connected by a single switch. Clusters connected by a single Ethernet switch are common in practice. Parts (b) and (c) of the figure show 32-node clusters of different logical topologies but the same physical topology, each having four switches with 8 nodes attached. Most current MPI implementations use a naive logical to physical topology mapping scheme. Both of the logical topologies can be easily created by having different host files. The three topologies are referred to as *topology (a)*, *topology (b)*, and *topology (c)*.

The performance of the tuned routines is compared with routines in LAM/MPI 6.5.9 and a recently improved MPICH 1.2.6 [77] using both micro-benchmarks and applications. The tuned routines are built on LAM point-to-point primitives. During the experiments, MPICH point-to-point primitives were not as efficient as LAM on the experimental platform. As a result, even though MPICH has more advanced collective communication algorithms in comparison to LAM, it does not always achieve higher performance. To make a fair comparison, MPICH-1.2.6 routines are ported to LAM. MPICH-LAM represents the ported routines. The term TUNED denotes the tuned routines. In the evaluation, TUNED is compared with LAM, MPICH, and MPICH-LAM.

Table 4.1: Tuned MPI_Alltoall, MPI_Allgather, and MPI_Allreduce

topo.	MPI_Alltoall	MPI_Allgather	MPI_Allreduce
(a)	*simple* ($n < 8718$) *pair light barrier* ($n < 31718$) *pair N MPI barriers* ($n < 72032$) *pair MPI barrier* (else)	*2D mesh* ($n < 10844$) *TSLR* ($n < 75968$) *pair MPI barrier* (else)	*tuned all-gather* ($n < 112$) *rdb* ($n < 9468$) *Rab1* ($n < 60032$) *MPICH Rab* ($n < 159468$) *Rab2* (else)
(b)	*bruck* ($n < 112$) *ring* ($n < 3844$) *ring N MPI barriers* ($n < 6532$) *ring light barrier* ($n < 9968$) *pair N MPI barriers* ($n < 68032$) *pair MPI barrier* (else)	*3D mesh* ($1n < 208$) *TSLR* (else)	*Rab1* ($n < 17$) *rdb* ($n < 395$) *MPICH Rab* ($n < 81094$) *Rab2* (else)
(c)	*bruck* ($n < 86$) *simple* ($n < 14251$) *pair MPI barrier* (else)	*3D mesh* ($n < 3999$) *TSLR* (else)	*tuned all-gather* ($n < 17$) *rdb* ($n < 489$) *Rab1* ($n < 20218$) *Rab2* (else)

4.1.2.1 Tuned Routines & Tuning Time

Table 4.1 shows the tuned *MPI_Alltoall*, *MPI_Allgather*, and *MPI_Allreduce* for topologies (a), (b), and (c). In this table, the algorithms selected in the tuned routines are sorted in increasing order based on their applicability to message sizes. This approach allows for using the message size upper bound to specify the range of the message sizes that an algorithm is applied in the tuned routine. For example, the pair with light barrier algorithm is applied in the tuned *MPI_Alltoall* on topology (a) for message sizes from 8718 bytes to 31717 bytes. For comparison, the algorithms in LAM/MPI and MPICH are depicted in Table 4.2. Since topologies (a), (b), and (c) have either 16 nodes or 32 nodes, only the algorithms for 16 nodes or 32 nodes are included in Table 4.2. There are a number of important observations. First, from Table 4.1, it can be seen that for different topologies, the best performing algorithms for each operation are quite different, which indicates that the one-scheme-fits-all approach in MPICH and LAM cannot achieve good performance for different topologies. Second, the topology-specific algorithms are part of the tuned *MPI_Allgather* and *MPI_Allreduce* routines for all three topologies. Although the topology specific all-to-all routine is not selected in the tuned routines for the three topologies, it offers the best performance for other topologies when the message size is large. These results indicate that using topology-unaware algorithms alone is insufficient to obtain high performance routines. Hence, an empirical approach must be used with the topology-specific routines to construct efficient communication routines for different topologies. Third, although the MPICH algorithms in general are much better than LAM algorithms, in many cases, they do not use the best algorithms for the particular topology and for the particular message size. As will be shown shortly, by empirically selecting better algorithms, the tuned routines sometimes out-perform MPICH routines to a very large degree.

Table 4.2: LAM/MPI and MPICH algorithms for MPI_Alltoall, MPI_Allgather, and MPI_Allreduce

routine	LAM	MPICH
MPI_Alltoall	*simple*	*bruck* ($n \leq 256$) *spreading simple* ($n \leq 32768$) *pair* (else)
MPI_Allgather	*gather-bcast*	*rdb* ($n * p < 524288$) *LR* (else)
MPI_Allreduce	*reduce-bcast*	*rdb* ($n < 2048$) *MPICH Rab.* (else)

Table 4.3: Tuning time

tuned routine	topo. (a)	topo. (b)	topo. (c)
MPI_Alltoall	1040s	6298s	6295s
MPI_Allgather	1157s	6288s	6326s
MPI_Allreduce	311s	261s	296s
MPI_Alltoallv	64s	177s	149s
MPI_Allgatherv	63s	101s	112s

Table 4.3 shows the tuning time of STAGE-MPI. In the table, the tuning time for *MPI_Allreduce* assumes that *MPI_Alltoall* and *MPI_Allgather* have been tuned. The time for *MPI_Alltoallv* is the tuning time for finding the best routine for one communication pattern: all–to–all with $1KB$ message size. The time for *MPI_Allgatherv* is the tuning time for finding the best routine for one communication pattern: all–gather with $1KB$ message size. The tuning time depends on many factors such as the number of algorithms to be considered, the

number of algorithms having parameters and the parameter space, the search heuristics, the network topology, and how the timing results are measured. As can be seen from the table, it takes minutes to hours to tune the routines. The time is in par with that for other empirical approach based systems such as ATLAS [82]. Hence, like other empirical based systems, STAGE-MPI is applicable when this tuning time is relatively insignificant, e.g. when the application has a long execution time, or when the application is executed repeatedly on the same platform.

4.1.2.2 Performance of Individual Routines

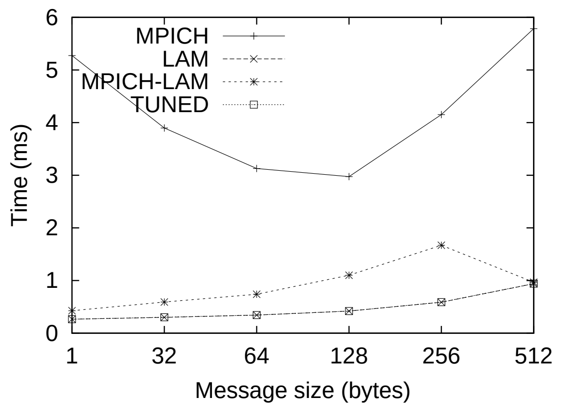
An approach similar to Mpptest [32], which is shown in Figure 4.3 (a), is used to measure the performance of an individual MPI routine. The number of iterations is varied according to the message size: more iterations are used for small message sizes to offset the clock inaccuracy. For the message ranges $1B - 3KB$, $4KB - 12KB$, $16KB - 96KB$, $128KB - 256KB$, and $> 256KB$, the respective number of iterations used are: 100, 50, 20, 10, and 5. The results for these micro-benchmarks are the averages of three executions. The *average* time among all nodes is used as the performance metric.

The results for *MPI_Alltoall*, *MPI_Allgather*, and *MPI_Allreduce* are reported. The performance of *MPI_Alltoallv* and *MPI_Allgatherv* depends on the communication pattern. The two routines will be evaluated with applications, and the results will be shown shortly. Since in most cases, MPICH has better algorithms than LAM, and MPICH-LAM offers the highest performance. the focus will be on comparing TUNED with MPICH-LAM. Before presenting the results, two general observations in the experiments are pointed out.

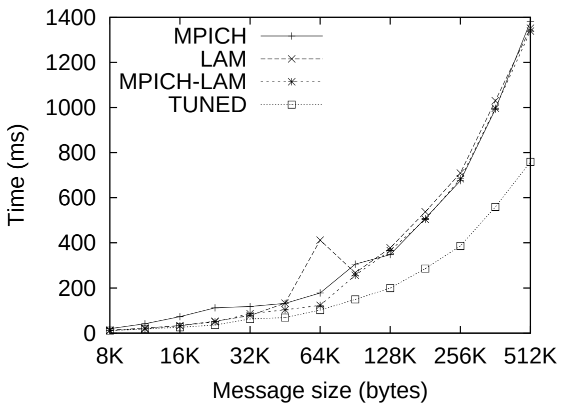
1. Ignoring the minor inaccuracy in the performance measurement, for all three topologies and all three operations, the tuned routines never perform worse than the best corresponding routines in LAM, MPICH, and MPICH-LAM.

2. For all three topologies and all three operations, the tuned routines out-perform the best corresponding routines in LAM, MPICH, and MPICH-LAM by at least 40% at some ranges of message sizes.

Figure 4.7 shows the performance of *MPI_Alltoall* results on topology (a). For small messages ($1 \leq n \leq 256$), both LAM and TUNED use the simple algorithm, which offers



(a) Small message sizes

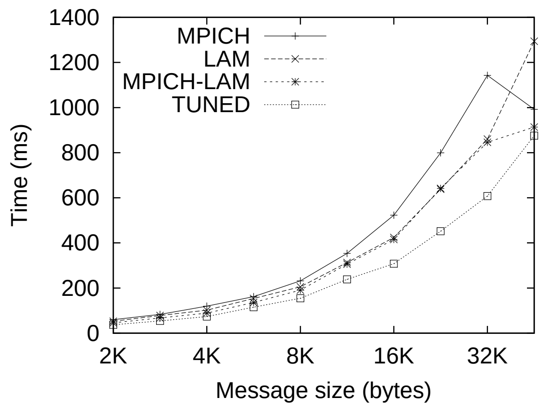


(b) Medium to large message sizes

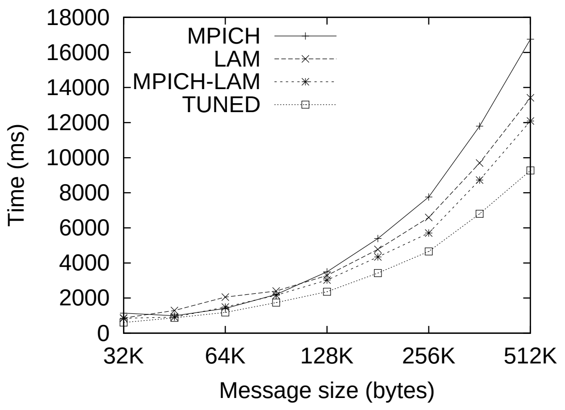
Figure 4.7: MPI_Alltoall on topology (a)

higher performance than the bruck algorithm used in MPICH. When the message size is 512 bytes, MPICH changes to the spreading simple algorithm, which has similar performance to the simple algorithm. TUNED, LAM, and MPICH-LAM have similar performance for the message size in the range from 256 bytes to 9K bytes. Figure 4.7 (b) shows the results for larger message sizes. For large messages, TUNED offers much higher performance than both MPICH and LAM. For example, when the message size is $128KB$, the time for TUNED is 200.1ms and the time for MPICH-LAM (the best among LAM, MPICH, and MPICH-LAM) is 366.2ms, which constitutes an 83% speedup. The performance curves for topology (b) and topology (c) show a similar trend. Figure 4.8 shows the results for topology (b). For a very wide range of message sizes, TUNED is around 20% to 42% better than the best among LAM, MPICH, and MPICH-LAM.

Figure 4.9 shows the performance results for *MPI_Allgather* on topology (c). When the message size is small, TUNED performs slightly better than other libraries. However, when the message size is large, the tuned routine significantly out-performs routines in other libraries. For example, when the message size is $32KB$, the time is 102.5ms for TUNED, 1362ms for LAM, 834.9ms for MPICH, and 807.9ms for MPICH-LAM. TUNED is about 8 times faster than MPICH-LAM. This demonstrates how much performance differences can be made when the topology information is taken into consideration. In fact, the topology-specific logical ring algorithm (TSLR), used in TUNED, can in theory achieve the same

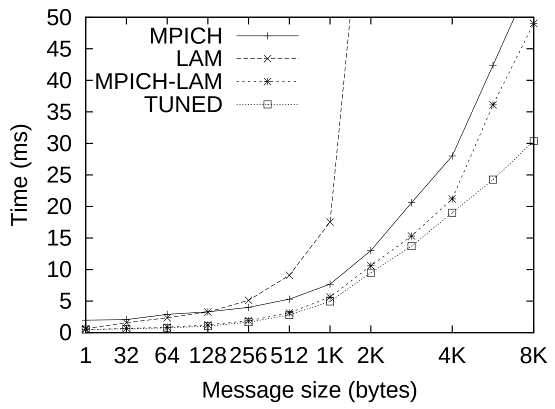


(a) Medium message sizes

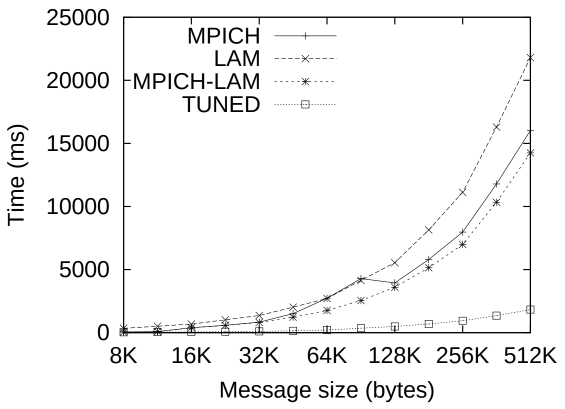


(b) Large message sizes

Figure 4.8: MPI_Alltoall on topology (b)



(a) Small message sizes

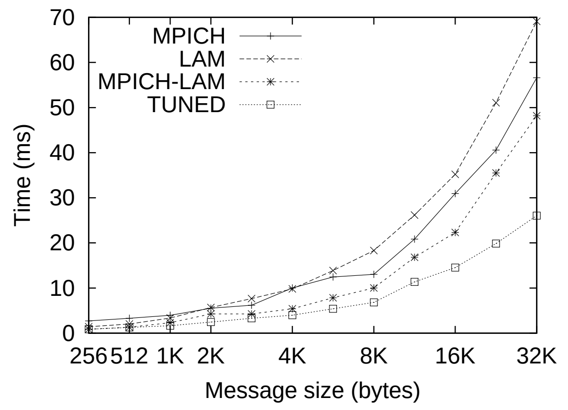


(b) Medium to large message sizes

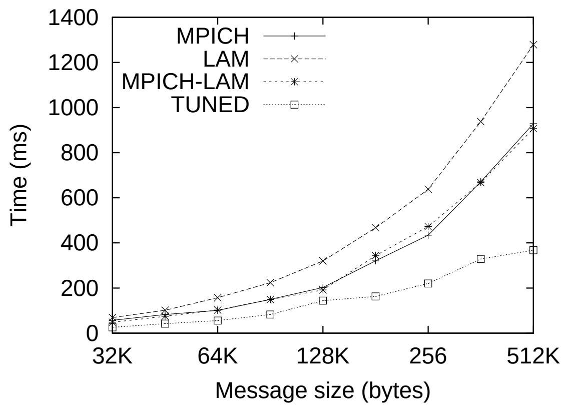
Figure 4.9: MPI_Allgather on topology (c)

performance for any Ethernet switched cluster with any number of nodes as the performance for a cluster with the same number of nodes connected by a single switch. On the other hand, the performance of the topology-unaware logical ring algorithm (LR), used in MPICH, can be significantly affected by the way the logical nodes are organized.

Figure 4.10 shows the results for *MPI_Allreduce* on topology (c). TUNED and MPICH-LAM have a similar performance when the message size is less than 489 bytes. When the message size is larger, TUNED out-performs MPICH-LAM to a very large degree even



(a) Small to medium message sizes



(b) Medium to large message sizes

Figure 4.10: MPI_Allreduce on topology (c)

though for a large range of message sizes, both TUNED and MPICH-LAM use variations of the Rabenseifner algorithm. For example, for message size 2048 bytes, the time is 2.5ms for TUNED versus 4.3 ms for MPICH-LAM. For message size $64KB$, the time is 55.9ms for TUNED versus 102.4ms for MPICH-LAM.

4.1.2.3 Performance of Application Programs

Three application programs are used in the evaluation: IS, FT, and NTUBE. IS and FT come from the *Nas Parallel Benchmarks* (NPB) [54]. The IS (Integer Sort) benchmark sorts N keys in parallel and the FT (Fast Fourier Transform) benchmark solves a partial differential equation (PDE) using forward and inverse FFTs. Both IS and FT are communication intensive programs with most communications performed by *MPI_Alltoall* and *MPI_Alltoallv* routines. The class B problem size supplied by the benchmark suite is used for the evaluation. The NTUBE (Nanotube) program performs molecular dynamics calculations of thermal properties of diamond [66]. The program simulates 1600 atoms for 1000 steps. This is also a communication intensive program with most communications performed by *MPI_Allgatherv*.

Table 4.4 shows the execution time for using different libraries with different topologies. The tuned library consistently achieves much better performance than the other implementations for all three topologies and for all programs. For example, on topology (a), TUNED improves the IS performance by 59.8% against LAM, 338.1% against MPICH, and 61.9%

Table 4.4: Execution time (seconds)

benchmark	library	topo. (a)	topo. (b)	topo. (c)
IS	LAM	15.5s	38.4s	36.5s
	MPICH	42.5s	58.2s	51.5s
	MPICH-LAM	15.7s	35.5s	33.4s
	TUNED	9.7s	28.4s	28.6s
FT	LAM	409.4s	320.8s	281.4s
	MPICH	243.3s	365.8s	281.1s
	MPICH-LAM	242.0s	246.0s	305.6s
	TUNED	197.7s	206.0s	209.8s
NTUBE	LAM	214.3s	304.1s	179.6s
	MPICH	49.7s	244.5s	88.7s
	MPICH-LAM	47.2s	236.8s	80.9s
	TUNED	35.8s	47.6s	45.0s

against MPICH-LAM. Notice that the execution time on topologies (b) and (c) is larger than that on topology (a) even though there are 32 nodes on topologies (b) and (c) and 16 nodes on topology (a). This is because all programs are communication bounded and the network in topologies (b) and (c) has a smaller aggregate throughput than that in topology (a).

4.1.2.4 Impacts of Different Timing Mechanisms on Performance of Collective Routines

As mentioned previously, STAGE-MPI supports three built-in timing mechanisms that can be used to measure the performance of collective routines. Thus, it is worthwhile studying the impacts of using a different performance measurement scheme on the performance of collective routines. The study is performed as follows. STAGE-MPI tunes the collective routines using different timing mechanisms, including *Mpptest, Mpptest + barrier,* and *Mpptest + computation* (with a computation load introducing 25ms in each iteration). The final generated routines are used in the application benchmarks, and the performance is measured.

Table 4.5 summarizes the results of using different timing mechanisms on the performance of communication routines in the application benchmarks. For each application benchmark,

Table 4.5: Performance of applications on topology (a) using collective routines tuned with different timing mechanisms

benchmark	timing scheme	communication algorithm	time
IS MPI_Alltoallv	Mpptest	*ring with MPI barrier*	9.70s
	Mpptest+barrier	*ring with light barrier*	9.79s
	Mpptest+computation	*ring with MPI barrier*	9.70s
FT MPI_Alltoall	Mpptest	*ring with MPI barrier*	197.7s
	Mpptest+barrier	*ring with light barrier*	192.3s
	Mpptest+computation	*ring with MPI barrier*	197.7s
NTUBE MPI_Allgatherv	Mpptest	*rdb*	35.80s
	Mpptest+barrier	*spreading simple*	34.79s
	Mpptest+computation	*LR*	35.72s

the table shows the communication algorithms that resulted from STAGE-MPI using different timing mechanisms as well as the total application execution time. The results shown are for topology (a). When conducting the study on topologies (b) and (c), it was observed that the different timing schemes resulted in the same communication algorithms for the particular collective routines used in the benchmarks. There are two observations that can be drawn from the table. First, the performance of the applications with collective routines tuned with different performance measurement schemes is quite similar. Second, while in some cases tuning collective routines using different timing mechanisms results in the same communication algorithms, there are cases where different timing mechanisms can result in the selection of different communication algorithms. For example, tuning the all-gatherv operation on topology (a) for the NTUBE benchmark with the three timing mechanisms results in three different communication algorithms. However, the selected algorithms under the different timing scheme seem to have similar performance, which is reflected in the similar total execution times.

The results of this preliminary study shows that using any of the three built-in timing mechanisms in STAGE-MPI to tune the collective routines is very likely to yield routines of similar performance. While this is observed in the set of benchmarks and on the topologies considered in this study, investigating this issue further is an interesting future direction.

4.2 Application Behavior and Its Impacts on Collective Operations

Although STAGE-MPI results in an implementation of MPI collective communication routines with a level of software adaptability exceeding that of LAM/MPI [44] and MPICH [52], it still has some limitations. One limitation is that the generated routines can only adapt to the platform when the standard Mpptest micro-benchmark [32] is used to measure the performance. In any MPI application program, collective communication routines can be used in different program contexts, and in every context, the application behaves differently. Different application behavior can have a significant impact on the performance of a collective communication algorithm, which indicates that a different algorithm may need to be used for a different context in order to achieve high performance. This section investigates application behavior as well as its impacts on the MPI collective communication operations. The results indicate that the application behavior has a significant impact on performance. This implies that STAGE-MPI routines may not be optimal. Ideally, a DF library should be able to adapt to both the platform and the application.

For an MPI collective operation, the application behavior is summarized as the *process arrival pattern*, which defines, as a result of involving multiple processes in a collective operation, the timing when different processes arrive at the collective operation (the call site of the collective communication routine). A process arrival pattern is said to be *balanced* when all processes arrive at the call site at roughly the same time, and *imbalanced* otherwise. The terms, balanced and imbalanced process arrival patterns, are quantified in Section 4.2.1.

The process arrival pattern (application behavior) can have a profound impact on the performance of a collective operation because it decides the time when each process can start participating in the operation. Unfortunately, this important factor has been largely overlooked. MPI developers routinely make the implicit assumption that all processes arrive at the same time (a balanced process arrival pattern) when developing and analyzing algorithms for MPI collective communication operations [61, 77]. In fact, it is widely believed that, to develop a high performance MPI application, application developers should balance the computation and let MPI developers focus on optimizing the cases with balanced process arrival patterns. It will be shown later that this common belief has flaws: even assuming that the computation in an application is perfectly balanced, the process arrival patterns for

MPI collective operations are usually imbalanced.

In this section, the process arrival patterns for MPI collective operations are studied in a number of applications. The study [21] is performed on two commercial off-the-shelf (COTS) clusters: a high-end Alphaserver cluster (the Lemieux machine located in Pittsburgh Supercomputing Center (PSC) [60]) and a low-end Beowulf cluster with Gigabit Ethernet connection. These two clusters are believed to be representative, and that the results can apply to a wide range of practical clusters. In this study, the process arrival patterns in a set of MPI benchmarks are characterized on these two clusters, a micro-benchmark where each process performs exactly the same computation is examined to understand the behavior of applications with a balanced computation load, and the impacts of imbalanced process arrival patterns on some commonly used algorithms for MPI collective operations are investigated. The findings of this study include the following.

- The process arrival patterns for MPI collective operations are usually imbalanced. In all benchmarks, the balanced process arrival patterns are only observed for collective operations that follow other collective operations (consecutive collective calls).

- In cluster environments, it is virtually impossible for application developers to control the process arrival patterns in their programs without explicitly invoking a global synchronization operation.

- The performance of the MPI collective communication algorithms that were studied is sensitive to process arrival patterns. In particular, the algorithms that perform better with a balanced process arrival pattern tend to perform worse when the process arrival pattern becomes more imbalanced.

This section is organized as follows. First, the process arrival pattern and the parameters used to characterize it are formally defined. Second, the benchmarks are described, the experimental setup and the data collection method are explained, and the statistics of process arrival patterns in the programs are presented. Third, the process arrival patterns in a micro-benchmark that has a perfect computation load distribution is presented and the causes for such a program to have imbalanced process arrival patterns are investigated. Fourth, the impact of process arrival patterns on some common algorithms for MPI collective operations is evaluated. Finally, the implication of the imbalanced process arrival patterns on the development and evaluation of algorithms for MPI collective operations are discussed.

4.2.1 Process Arrival Pattern

Let n processes, p_0, p_1, ..., p_{n-1}, participate in a collective operation. Let a_i be the time that process p_i arrives at the collective operation. The *process arrival pattern* can be represented by the tuple $(a_0, a_1, ..., a_{n-1})$. The average process arrival time is $\bar{a} = \frac{a_0+a_1+...+a_{n-1}}{n}$. Let f_i be the time that process p_i finishes the operation. The *process exit pattern* can be represented by the tuple $(f_0, f_1, ..., f_{n-1})$. The elapsed time that process p_i spends in the operation is thus $e_i = f_i - a_i$, the total time is $e_0 + e_1 + ... + e_{n-1}$, and the average per node time is $\bar{e} = \frac{e_0+e_1+...+e_{n-1}}{n}$. In an application, the total time or the average per node time accurately reflects the time that the program spends on the operation. The average per node time ($\bar{e}$) is used to denote the performance of an operation (or an algorithm).

Let us use the term **imbalance** in the process arrival pattern to signify the differences in the process arrival times at a collective communication call site. Let δ_i be the time difference between p_i arrival time a_i and the average arrival time $\bar{a}$, $\delta_i = |a_i - \bar{a}|$. The imbalance in the process arrival pattern can be characterized by the *average case imbalance time*, $\bar{\delta} = \frac{\delta_0+\delta_1+...+\delta_{n-1}}{n}$, and the *worst case imbalance time*, $\omega = max_i\{a_i\} - min_i\{a_i\}$. Figure 4.11 shows an example process arrival pattern on a 4-process system and depicts the parameters described.

An MPI collective communication operation typically requires each process sending multiple messages to other processes. A collective communication algorithm organizes the messages in the operation in a certain way. For example, in the *pair* algorithm for *MPI_Alltoall* [77], the messages in the all-to-all operation are organized in $n - 1$ phases: in phase $0 \leq i \leq n - 1$, process p_j sends a message to process $p_{j \oplus i}$ (exclusive or) and receives a message from the same process. The impact of an imbalance process arrival pattern on the performance of a collective algorithm is mainly caused by the early completions or late starts of some messages in the operation. In the *pair* algorithm, early arrivals of some processes will cause some processes to complete a phase and start the next phase while other processes are still in the previous phase, which may cause system contention and degrade the performance. Hence, the impacts of an imbalanced process arrival pattern can be better characterized by the number of messages that can be completed during the period when some processes arrive

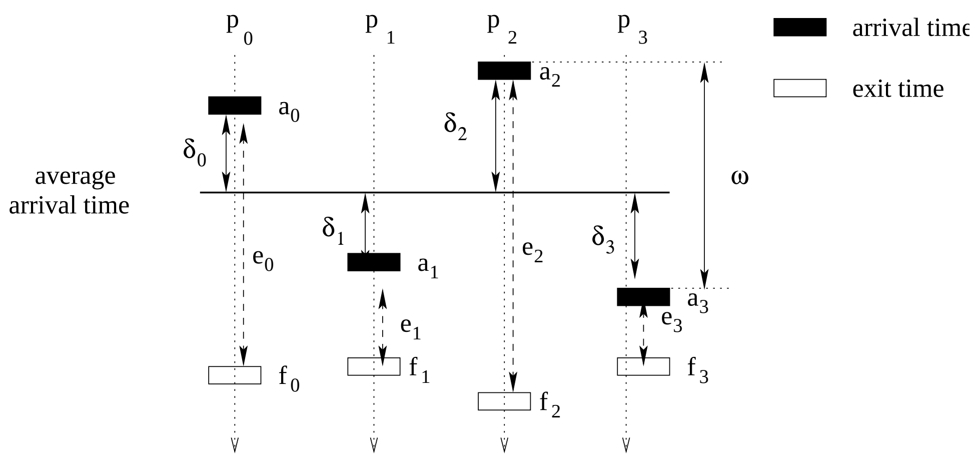


Figure 4.11: Process arrival pattern

while others do not. To capture this notion, the worst case and average case imbalance times are normalized by the time to communicate one message. The normalized results are called the *average/worst case imbalance factor*. Let T be the time to communicate one message in the operation, the *average case imbalance factor* equals to $\frac{\bar{\delta}}{T}$ and the *worst case imbalance factor* equals to $\frac{\omega}{T}$. A worst case imbalance factor of 2 means that by the time the last process arrives at the operation, the process that arrives first may have completed two messages in the operation. The imbalanced process arrival patterns will be characterized by their average and worst case imbalance factors. In general, a process arrival pattern is **balanced** if the worst case imbalance factor is less than 1 (all processes arrive within a message time) and **imbalanced** otherwise.

4.2.2 Process Arrival Patterns in MPI Programs on Two Platforms

In the following, a brief description of the platforms, benchmarks, and data collection method used in this study is presented. The process arrival pattern statistics are then discussed.

4.2.2.1 Platforms

The process arrival pattern statistics are collected on two representative platforms: a high-end cluster and a low-end Beowulf cluster. The high-end cluster is the Lemieux machine located in Pittsburgh Supercomputing Center (PSC) [60]. The machine consists of 750 Compaq Alphaserver ES45 nodes, each of which includes four 1-GHz SMP processors with 4GB of memory. The nodes are connected with a Quadrics interconnection network, and they run Tru64 Unix operating system. All programs are compiled with the native *mpicc* on the system and linked with the native MPI and ELAN library. ELAN is a low-level internode communication library for Quadrics. Some of the times for point-to-point communications with different message sizes between processors in different nodes are summarized in Table 4.6. These numbers, which are obtained using a pingpong program, are used to compute imbalance factors in the benchmark study.

The low-end cluster is a 16-node Beowulf cluster. The nodes of the cluster are Dell Dimension 2400 with a 2.8GHz P4 processor, 128MB of memory, and 40GB of disk space. All machines run Linux (Fedora) with 2.6.5-1.358 kernel. The Ethernet card in each machine is Broadcom BCM 5705 with the driver from Broadcom. These machines are connected to a Dell PowerConnect 2624 1Gbps Ethernet switch. This system uses MPICH 2-1.0.1 for communication. All programs are compiled with the *mpicc* that comes with the MPICH package. Some of the times for point-to-point communications with different message sizes are summarized in Table 4.7.

Table 4.6: One way point-to-point communication time on Lemieux

size (B)	time (ms)	size (B)	time (ms)	size (B)	time (ms)
4	0.008	4K	0.029	64K	0.291
256	0.008	16K	0.079	128K	0.575
1024	0.0207	32K	0.150	256K	1.138

Table 4.7: One way point-to-point communication time on the Beowulf cluster

size (B)	time (ms)	size (B)	time (ms)	size (B)	time (ms)
4	0.056	4K	0.150	64K	0.846
256	0.063	16K	0.277	128K	1.571
1024	0.088	32K	0.470	256K	3.120

4.2.2.2 Benchmarks

Table 4.8: Summary of benchmarks (e. time is execution time while c. time is communication time)

benchmark	description	#lines	e. time (n=64)	c. time (n=64)
FT	solves PDE with forward and inverse FFTs	2234	13.4s	8.3s
IS	sorts integer keys in parallel	1091	2.2s	1.6s
LAMMPS	simulates dynamics of molecules in different states	23510	286.7s	36.1s
PARADYN	simulates dynamics of metals and metal alloys molecules	6252	36.6s	33.1s
NBODY	simulates effects of gravitational forces on N bodies	256	59.5s	1.5s
NTUBE 1	performs molecular dynamics calculations of diamond	4480	894.4s	32.3s
NTUBE 2	performs molecular dynamics calculations of diamond	4570	852.9s	414.1s

Table 4.8 summarizes the seven benchmarks. For reference, the code size is shown as well as the execution and collective communication elapsed times for running the programs on 64 processors on Lemieux. Table 4.9 shows the major collective communication routines in the benchmarks and their dynamic counts and message sizes assuming the number of processors is 64. There are significant collective operations in all programs. Next, each benchmark and the related parameters/settings used in the experiments are briefly described.

FT (Fast-Fourier Transform) is one of the parallel kernels included in the NAS parallel benchmarks [54]. FT solves a partial differential equation (PDE) using forward and inverse FFTs. The collective communication routines used in this benchmark include *MPI_Alltoall*, *MPI_Barrier*, *MPI_Bcast*, and *MPI_Reduce* with most communications being carried out by the *MPI_Alltoall* routine. Class B problem size supplied by the NAS benchmark suite is used in the evaluation.

IS (Integer Sort) is a parallel kernel from the NAS parallel benchmarks. It uses bucket sort to order a list of integers. The MPI collective routines in this program are *MPI_Alltoall*, *MPI_Alltoallv*, *MPI_Allreduce*, and *MPI_Barrier*. Most communications in this program are carried out by the *MPI_Alltoallv* routine. Class B problem size for this benchmark is used in the experiments.

LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) [45] is a classical parallel molecular dynamics code. It models the assembly of particles in a liquid, solid, or gaseous state. The code uses *MPI_Allreduce*, *MPI_Bcast*, and *MPI_Barrier*. The program is run with 1720 copper atoms for 3000 iterations in the experiments.

Table 4.9: The dynamic counts of major collective communication routines in the benchmarks ($n = 64$)

benchmark	routine	msg size	dynamic count
FT	MPI_Alltoall	131076	22
	MPI_Reduce	16	20
IS	MPI_Alltoallv	33193*	11
	MPI_Allreduce	4166	11
	MPI_Alltoall	4	11
LAMMPS	MPI_Allreduce	42392	2012
	MPI_Bcast	4-704	48779
	MPI_Barrier		4055
PARADYN	MPI_Allgatherv	6-1290*	16188
	MPI_Allreduce	4-48	13405
NBODY	MPI_Allgather	5000	300
NTUBE 1	MPI_Allgatherv	16000*	1000
NTUBE 2	MPI_Allreduce	8	1000

* the average of all message sizes in the v-version routines.

PARADYN (Parallel Dynamo) [58] is a classical parallel molecular dynamics simulation. It utilizes the embedded atom method potentials to model metals and metal alloys. The program uses *MPI_Allgather*, *MPI_Allgatherv*, *MPI_Allreduce*, *MPI_Bcast*, and *MPI_Barrier*. In the experiments, an example input file supplied by the benchmark code is used and 6750 atoms of liquid crystals are simulated in 1000 time steps.

NBODY [57] simulates over time steps the interaction, in terms of movements, positions and other attributes, among the bodies as a result of the net gravitational forces exerted on one another. It has applications in various areas such as strophysics, molecular dynamics and plasma physics. The code is a naive implementation of the *nbody* method and uses *MPI_Allgather* and *MPI_Gather* collective communications. The code runs with 8000 bodies and for 300 time steps.

NTUBE 1 performs molecular dynamics calculations of thermal properties of diamond [66]. This version of the code uses *MPI_Allgatherv* and *MPI_Reduce*. In the evaluation, the program runs for 1000 steps and each processor maintains 100 atoms.

NTUBE 2 is a different implementation of the Nanotube program. The functionality of NTUBE 2 is exactly the same as NTUBE 1. The collective communication routines used in this program are *MPI_Allreduce* and *MPI_Reduce*. In the evaluation, the program runs for 1000 steps with each processor maintaining 100 atoms.

4.2.2.3 Data Collection

To investigate process arrival patterns and other statistics of MPI collective communications, an MPI wrapper library was developed. The wrapper records an event at each MPI process for each entrance and exit of an MPI collective communication routine. An event record contains information about the timing, the operation, and the message size of the collective operation. The times are measured using *MPI_Wtime* routine. Events are stored in memory during program execution. Once the application code calls *MPI_Finalize*, all processors write these events to a log file for post-mortem analysis. On Lemieux, the experiments are conducted with a batch partition of 32, 64, and 128 processors (4 processors per node).

4.2.2.4 Process Arrival Pattern Statistics

Table 4.10 and Table 4.11 show the average of the worst/average case imbalance factors among all collective routines in each benchmark on Lemieux and Beowulf cluster respectively. The two tables reveal several notable observations.

Table 4.10: The average of the worst/average case imbalance factors among all collective routines on Lemieux

benchmark	imbalance factor					
	n = 32		n = 64		n = 128	
	average	worst	average	worst	average	worst
FT	1017	6751	215	1266	91	633
IS	102	521	80	469	61	357
LAMMPS	9.1	40	6.1	33	3.8	23
PARADYN	5.4	27	5.3	28	8.7	44
NBODY	17	90	15	104	13	129
NTUBE 1	572	2461	2281	16K	4524	36K
NTUNE 2	24K	85K	44K	168K	83K	336K

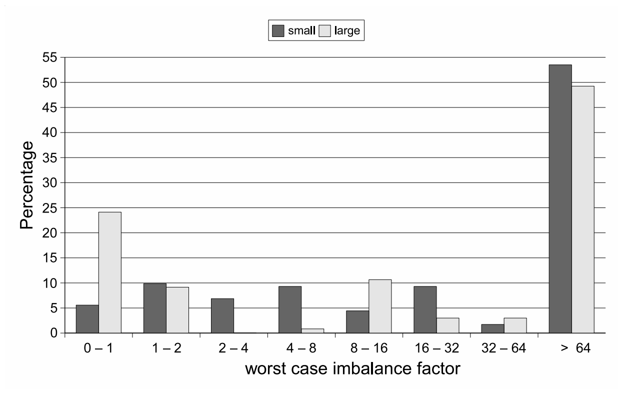
First, the average of the worst case imbalance factor for all programs on both clusters are quite large, even for FT, whose computation is fairly balanced. Second, the process arrival pattern depends heavily on the system architecture. For example, the imbalance factors for

Table 4.11: The average of the worst/average case imbalance factors among all collective routines on the Beowulf cluster

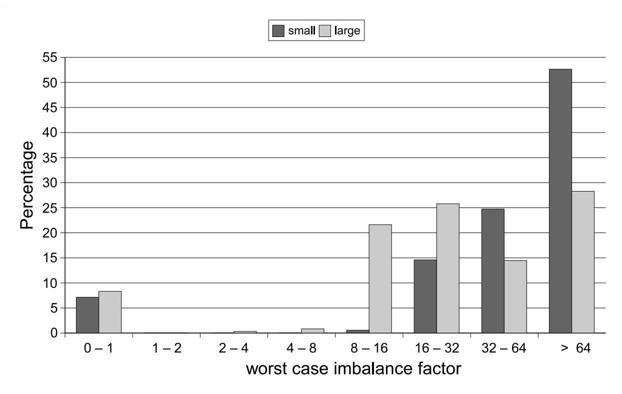
benchmark	imbalance factor	
	average	worst
FT	256	1119
IS	1241	9536
LAMMPS	252	582
PARADYN	11	73
NBODY	12	48
NTUBE 1	4.1	17
NTUNE 2	8.2	36.1

NTUBE 1 and NTUBE 2 are much larger on Lemieux than on the Beowulf cluster. This is because these two programs were designed for single CPU systems. When running them on Lemieux, an SMP cluster, the process arrival patterns become extremely imbalanced. On the other hand, the imbalance factors for LAMMPS, which contains a large number of *MPI_Bcast* calls, are much smaller on Lemieux than on the Beowulf cluster. This is because Lemieux has hardware support for *MPI_Bcast* while the Beowulf cluster uses a point-to-point communication based implementation. It must be noted that the network on Lemieux is much faster than that on the Beowulf cluster: the same imbalance time will result in a larger imbalance factor on Lemieux. Third, the increase of the number of processors affects the imbalance factor (on Lemieux). On one hand, more processors may result in a larger worst case imbalance factor since more processors contribute to the worst case imbalance. This is usually the case when the per processor workload is fixed as in the NTUBE 1 benchmark. On the other hand, when the problem size is fixed as in FT and IS, the imbalance factors may decrease as the number of processors increases. This is due to the reduction of the processor and communication workload (and thus the imbalance) at each node as the number of processors increases.

Operations that account for most of the communication times typically have large message sizes. Thus, operations with large message sizes are distinguished from those with small message sizes in Figure 4.12, which shows the distribution of the worst case imbalance factor for both operations with medium and large message sizes (> 1000 bytes) and operations with small message sizes ($\leq$ 1000 bytes). All benchmarks are equally weighted when computing the distribution. The results on Lemieux are for the 128-processor case. As can be seen from parts (a) and (b) of the figure, there is a significant portion of operations with both small sizes and medium/large sizes having large imbalance factors ($>$ 32) and only a small fraction of the operations have a balanced process arrival pattern. For example, on Lemieux, only about 15% of the operations with small messages and 34% of the operations with medium and large messages have a worst case imbalance factor less than 2. The figure also shows that the process arrival patterns are more imbalanced on the Beowulf cluster than on Lemieux. On the 16-processor Beowulf cluster, more than 90% of process arrival patterns for both large and small message sizes have a worst case imbalance factor larger than 8, which means that when the last process arrives, the first process may have completed half of the operation.



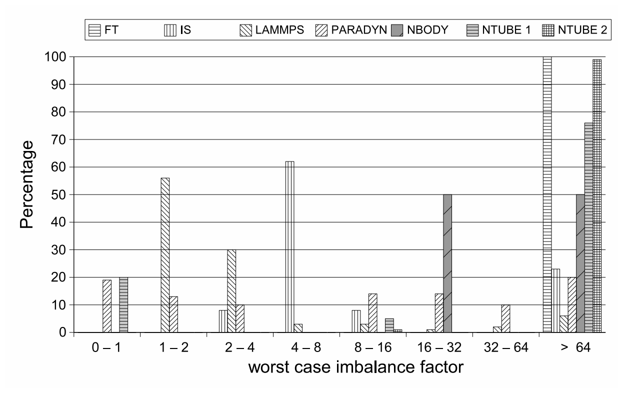
(a) Lemieux (128 processors)



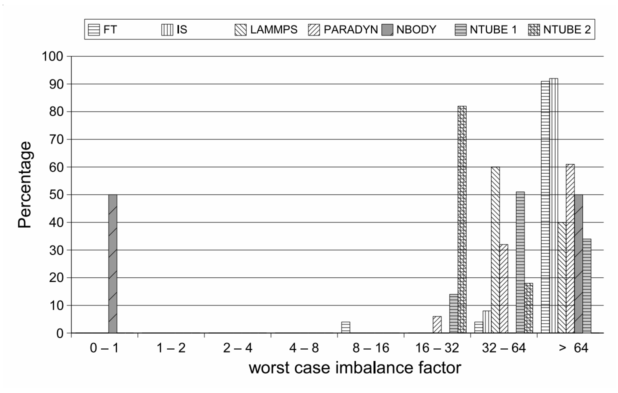
(b) Beowulf cluster

Figure 4.12: The distribution of worst case imbalance factors on the two platforms

Figure 4.13 and Figure 4.14 break down the distribution of the worst case imbalance factors in the seven benchmarks. Figure 4.13 shows the results for operations with small messages ($\leq$ 1000 bytes) on the two platforms while Figure 4.14 shows the results for operations with medium and large messages ($>$ 1000 bytes). From these figures, it can be seen that different benchmarks exhibit very different process arrival patterns. For example, on Lemieux, 100% of operations with medium/large messages in PARADYN have a balanced process arrival pattern (worst case balance factor $<$ 1) while a significant portion of the operations in other benchmarks have imbalanced process arrival patterns.



(a) Lemieux (128 processors)

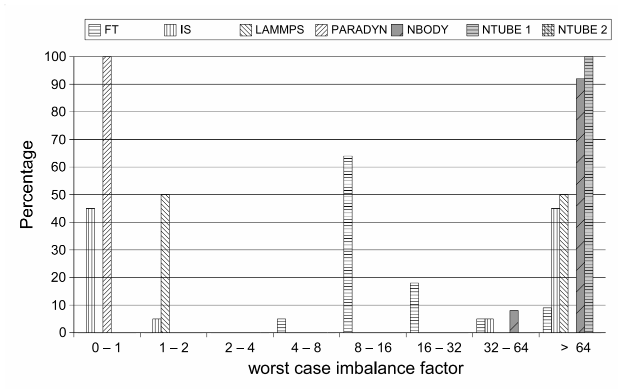


(b) Beowulf cluster

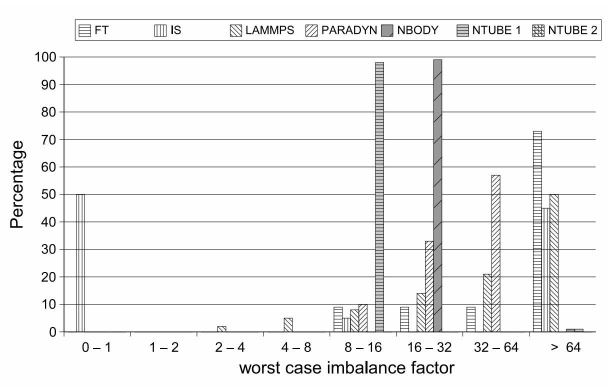
Figure 4.13: The distribution of worst case imbalance factors in each benchmark for operations with small message sizes ($\leq$ 1000 Bytes) on both platforms

While Figure 4.13 and Figure 4.14 give a good summary about the imbalance in the process arrival patterns in all benchmarks, not all summarized results are representative. For example, NBODY has a few operations with small messages, which are called in the application verification phase (outside the main loop). The imbalance factors for the collective operations that are important in the benchmarks (in the main loop and accounting for a significant amount of time) are shown in Table 4.12. Compared with the imbalance factors shown in Tables 4.10 and 4.11, it can be seen that the process arrival patterns for these important routines are generally more balanced than the average of all routines in the applications. This indicates that programmers are more careful about the load balancing issue in the main loop. However, on both platforms, only the *MPI_Alltoallv* in IS can be classified as having balanced process arrival patterns. Examining the source code reveals that this routine is called right after other MPI collective routines. All other routines have more or less imbalanced process arrival patterns.

Another interesting statistic is the characteristics of process arrival pattern for each individual call site. If the process arrival patterns for each call site in different invocations exhibit heavy fluctuation, the MPI routine for this call site must achieve high performance in all different types of process arrival patterns to be effective. On the other hand, if the process



(a) Lemieux (128 processors)



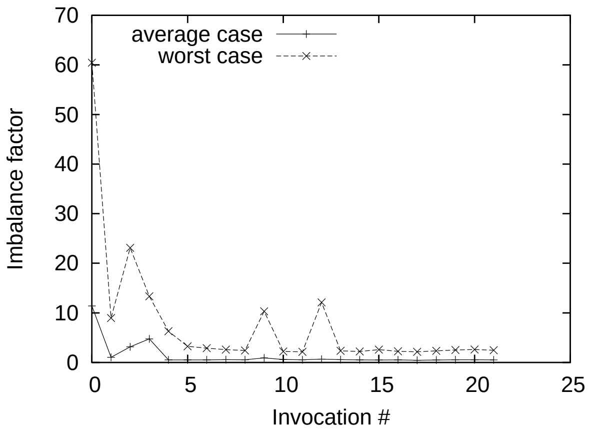
(b) Beowulf cluster

Figure 4.14: The distribution of worst case imbalance factors in each benchmark for operations with medium and large message sizes (> 1000 Bytes) on both platforms

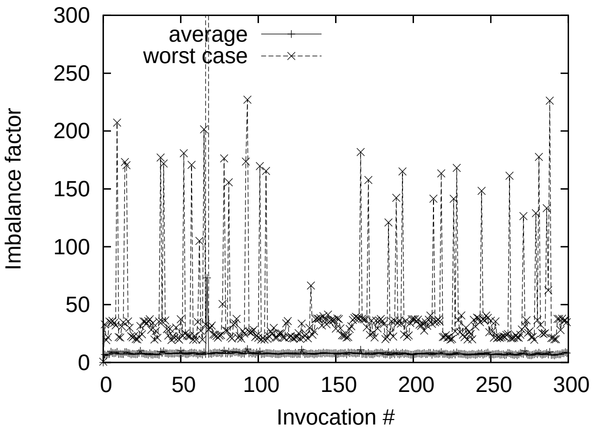
Table 4.12: The imbalance factor for major collective routines in the benchmarks

benchmark	major routine	Lemieux ($n = 128$)		Beowulf	
		imbalance factor		imbalance factor	
		average	worst	average	worst
FT	MPI_Alltoall	2.9	24	26	124
IS	MPI_Alltoallv	0.0	0.2	0.2	0.8
	MPI_Allreduce	148	772	3871	30K
LAMMPS	MPI_Bcast	0.2	3.2	276	620
	MPI_Allreduce	16	87.3	128	1201
	MPI_Barrier	40	222	98	411
PARADYN	MPI_Allgatherv	0.8	6	9.1	61
	MPI_Allreduce	15	68.7	13	86
NBODY	MPI_Allgather	13	129	12	48
NTUBE 1	MPI_Allgatherv	78	120	3.4	14
NTUBE 2	MPI_Allreduce	81K	314K	8.2	36

arrival patterns for the same call site is statistically similar, the MPI implementation will only need to optimize for the particular type of process arrival patterns, which is easier to accomplish. In the benchmarks, it is observed that the process arrival patterns for different invocations of the same call site exhibit a *phased* behavior: the process arrival patterns are statistically similar for a period of time before they change. In some cases, the process arrival patterns for the same call site are statistically similar in the whole program. Figure 4.15 depicts two representative cases on Lemieux with 128 processors. Part (a) of the figure shows the imbalance factors for each invocation of the *MPI_Alltoall* routine in FT while part (b) shows the imbalance factors for each invocation of the *MPI_Allgather* in NBODY. As can be seen from the figure, the majority of the calls have similar worst case and average case imbalance factors despite some large spikes that occur once in a while. This indicates that it might be feasible to customize the routine for each MPI call site and get good performance.



(a) MPI_Alltoall in FT



(b) MPI_Allgather in NBODY

Figure 4.15: The imbalance factors for major collective routines in FT and NBODY on Lemieux with 128 processors

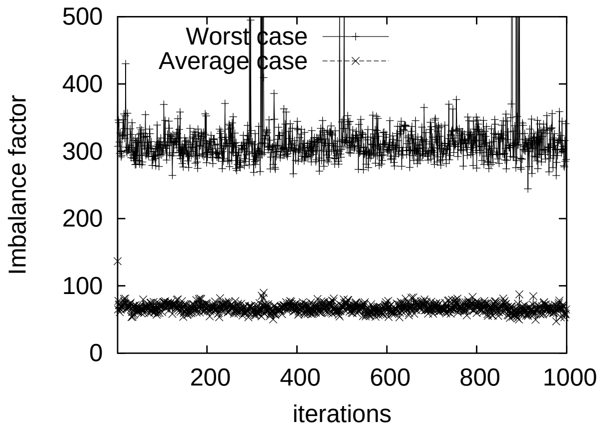
4.2.3 Process Arrival Patterns in a Micro-Benchmark

Since a well designed MPI program typically has a balanced computation load, understanding the process arrival patterns in this type of programs is particularly important. One surprising result in the previous subsection is that even programs with evenly distributed computation loads have very imbalanced process arrival patterns. However, these programs are too complex to determine what exactly is causing the imbalanced process arrival patterns. Let us study a simple micro-benchmark, shown in Figure 4.16, where all processes perform exactly the same amount of computation (computation loads are perfectly balanced), and measure the process arrival patterns. The goal is to determine whether application programmers can control the critical process arrival patterns in their MPI programs. In this micro-benchmark, a barrier is called before the main loop, which is executed 1000 times. There are two components inside the loop: lines (5) to (7) simulating the computation and an MPI_Alltoall() operation in line (9) after the computation. The computation time can be adjusted by changing the parameter XTIME.

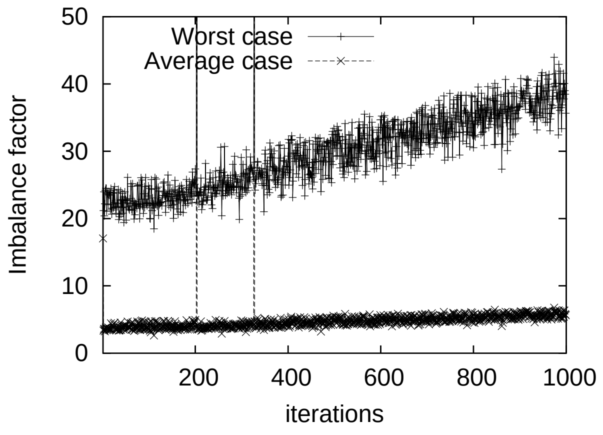
```
(1) ...
(2) MPI_Barrier(...);
(3) for (i=0; i<1000; i++) {
(4)    /* compute for roughly X milliseconds */
(5)    for (m=0; m< XTIME; m++)
(6)       for (k=1, k<1000; k++)
(7)          a[k] = b[k+1] - a[k-1] * 2;

(8)    arrive[i] = MPI_Wtime();
(9)    MPI_Alltoall(...);
(10)  leave[i] = MPI_Wtime()
(11)}
```

Figure 4.16: Code segment for a micro-benchmark

The process arrival patterns for the all-to-all operation are measured and the results for message size $64KB$ are shown. Smaller message sizes result in larger imbalance factors. The computation time is set to roughly 200ms for both clusters. Figure 4.17 shows the worst/average case imbalance factors in each iteration on Lemieux with 128 processors and on the 16-node Beowulf cluster. In both clusters, there is substantial imbalance in the process arrival patterns even though all processors perform exactly the same computations. The imbalance factors on Lemieux are larger than those on the Beowulf cluster for a few reasons. First, Lemieux has more processes and thus, has the higher chance to be imbalanced. Second, on Lemieux, different jobs share the network in the system; the uncertainty in messaging can cause the imbalance. Third, Lemieux has a faster network, the same imbalance time results in a higher imbalanced factor.



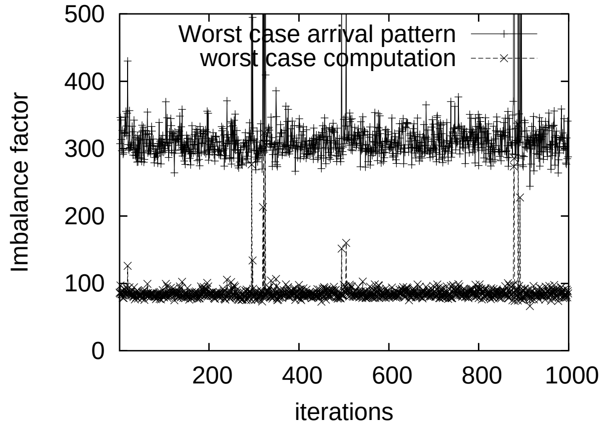
(a) Lemieux (128 processors)



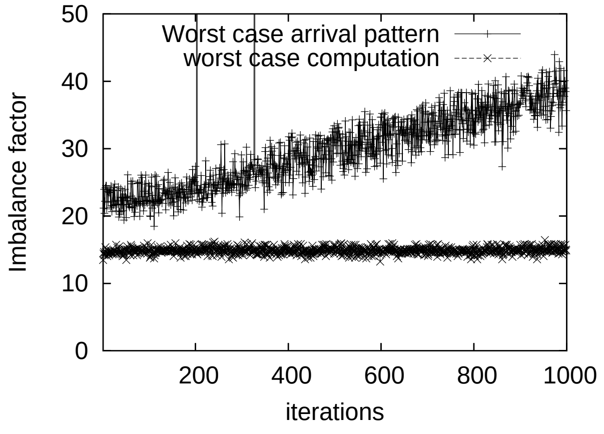
(b) Beowulf cluster

Figure 4.17: Process arrival patterns in the micro-benchmark (64KB message size, 200ms computation time) on the two platforms

One potential cause for the imbalanced process arrival patterns is that different machines may require different periods of time to run the computation even though the machines are homogeneous. An earlier study [59] has shown that this can be caused by operating system events. Figure 4.18 compares the computation imbalance in the micro-benchmark with the imbalance in process arrival patterns on the two platforms. The *computation imbalance time* is defined as the maximum time among all processes to execute the computation minus the minimum time among all processes. To consistently compare with the imbalance in arrival patterns, the computation imbalance time is normalized by the time to send one message ($64KB$). As can be seen in the figure, the computation imbalance is quite significant in both clusters. Yet, the worst case imbalance factor for the process arrival patterns is much larger than the computation imbalance. This is attributed to the fact that different processes take different times to perform the all-to-all operation: the imbalance in process arrival pattern is the accumulated effect of the imbalance in the computation and the imbalance in the process exit pattern of the all-to-all operations.



(a) Lemieux (128 processors)



(b) Beowulf cluster

Figure 4.18: Computation imbalance versus arrival pattern imbalance on the two platforms

To further understand the characteristics of the computation imbalance, the XTIME parameter is changed such that the computation lasts for 50ms, 100ms, 200ms, 400ms, and 800ms. The computation imbalance time in each iteration is measured, and the results for the average over the 1000 iterations are presented. Figure 4.19 plots the average computation imbalance time with respect to the total computation time on the two platforms. As can be seen from the figure, from 50ms to 800ms, the computation imbalance increases almost linearly with respect to the total computation time. This indicates that when the computation between two collective operation is sufficiently large, the process arrival pattern imbalance will also be very large. Moreover, this is inherent to the system, and it is impossible for application developers to overcome. This is the reason that, in the benchmark study of the previous subsection, only balanced process arrival patterns in consecutive collective routine calls were observed: when there is a computation in between, it is difficult to have a balanced process arrival pattern.

This study indicates that the way a program is coded is only one of many factors that can affect process arrival patterns. Other factors, including the inherent computation imbalance and the process exit pattern, which may be affected by the collective communication algorithm and network hardware, are beyond the control of application developers. It is impractical to assume that application programmers can balance the load to make the process

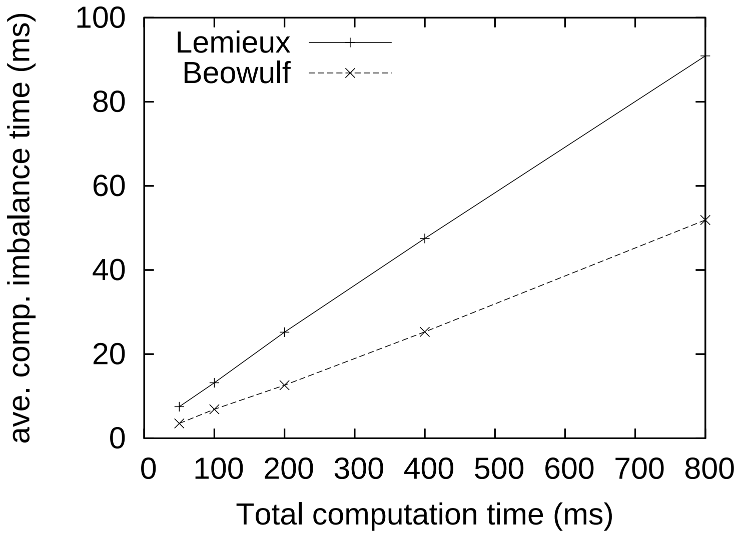


Figure 4.19: Computation imbalance versus total computation time

arrival patterns balanced in a program. The only way to ensure a balanced process arrival pattern is to explicitly call a synchronization operation. Hence, in most cases, the process arrival patterns in MPI programs will be imbalanced.

4.2.4 Impact of Imbalanced Process Arrival Patterns on the Performance of Collective Communication Algorithms

Most of the MPI collective communication algorithms were designed (and evaluated) assuming a balanced process arrival pattern. Since process arrival patterns in MPI programs are likely to be imbalanced, it would be interesting to know how these algorithms respond to imbalanced process arrival patterns. In this subsection, a study of the impact of the process arrival pattern on the performance of commonly used algorithms for *MPI_Alltoall* are reported.

Algorithms for *MPI_Alltoall* that are adopted in MPICH [77], including the *simple*, *bruck*, *pair*, and *ring* algorithms are considered. The *simple* algorithm basically posts all receives and all sends, starts the communications, and waits for all communications to finish. The order of communications for process p_i is $p_i \rightarrow p_{i+1}$, $p_i \rightarrow p_{i+2}$, ..., $p_i \rightarrow p_{(i+n-1)\ mod\ n}$. The *bruck* algorithm [7] is a $lg(n)$-step algorithm that is designed for achieving efficient all-to-all with small messages. The *pair* algorithm only works when the number of processes, n, is a power of two. It partitions the all-to-all communication into $n-1$ steps. In step i, process p_j sends and receives a message to and from process $p_{j \oplus i}$ (exclusive or). The *ring* algorithm also partitions the all-to-all communication into $n-1$ steps. In step i, process p_j sends a messages to process $p_{(j+i)\ mod\ n}$ and receives a message from process $p_{(j-i)\ mod\ n}$. All algorithms are implemented over MPI point-to-point primitives. More detailed description of these algorithms can be found in [77]. In addition to these algorithms, the native *MPI_Alltoall* provided on Lemieux, whose algorithm could not be identified, is also considered.

```
(1) r = rand() % MAX_IMBALANCE_FACTOR;
(2) for (i=0; i<ITER; i++) {
(3)   MPI_Barrier (...);
(4)   for (j=0; j<r; j++) {
(5)     ... /* computation time equal to one msg time */
(6)   }
(7)   t_0 = MPI_Wtime();
(8)   MPI_Alltoall(...);
(9)   elapse += MPI_Wtime() - t_0;
(10)}
```

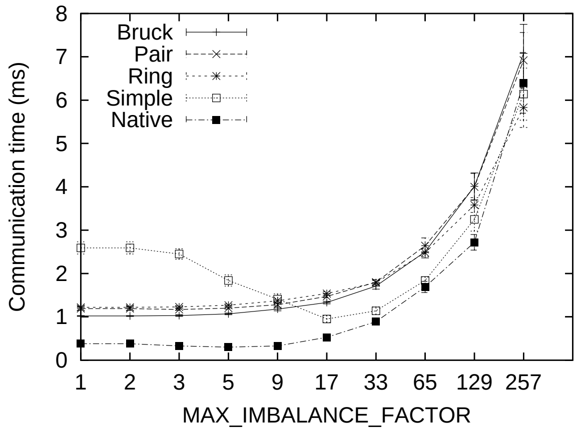
Figure 4.20: Code segment for measuring the impacts of imbalanced process arrival patterns

Figure 4.20 outlines the code segment that is used to measure the performance with a controlled imbalance factor in the random process arrival patterns. The worst-case imbalance factor is controlled by a variable *MAX_IMBALANCE_FACTOR*. Line (1) generates a random number, r, that is bounded by *MAX_IMBALANCE_FACTOR*. Before the all-to–all routine is measured (in lines (7) to (9)), the controlled imbalanced process arrival pattern is created by first calling a barrier (line (3)) and then introducing some computation between the barrier and the all-to-all routine. The time to complete the computation is controlled by r. The time spent in the loop body in line (5) is made roughly equal to the time for sending one message (see Table 4.6 and Table 4.7), and the total time for the computation is roughly equal to the time to send r messages. Hence, the larger the value of *MAX_IMBALANCE_FACTOR* is, the more imbalanced the process arrival pattern becomes. Note that the actual worst case imbalance factor, especially for small message sizes, may not be bounded by *MAX_IMBALANCE_FACTOR* since the process exit patterns of *MPI_Barrier* may not be balanced.

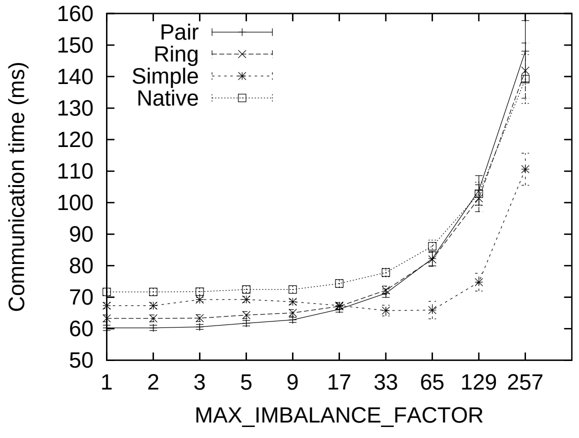
For each process arrival pattern, the routine is measured 100 times ($ITER = 100$) and the average elapsed time on each node is recorded. Each experiment involves one fixed value of *MAX_IMBALANCE_FACTOR*. For each experiment with one *MAX_IMBALANCE_FACTOR* value, 32 random experiments are performed (32 random process arrival patterns with the same *MAX_IMBALANCE_FACTOR*). The communication time for each experiment is reported by the confidence interval (of the average communication time) with a 95% confidence level, computed from the results of the 32 experiments.

Part (a) of Figure 4.21 shows the results for 1 Byte all-to-all communication on Lemieux with 32 processors. When $MAX\_IMBALANCE\_FACTOR \leq 9$, the bruck algorithm performs better than the ring and pair algorithms, and all three algorithms perform significantly better than the simple algorithm. However, when the imbalance factor is larger ($17 \leq MAX\_IMBALANCE\_FACTOR \leq 129$), the simple algorithm shows better results. The algorithm used in the native *MPI_Alltoall* routine performs much better than all four algorithms in the case when $MAX\_IMBALANCE\_FACTOR \leq 129$. When $MAX\_IMBALANCE\_FACTOR = 257$, the native algorithm performs worse than the ring and simple algorithms. These results show that under different process arrival patterns with different worst case imbalance factors, the algorithms have very different performance. When the imbalance factor increases, one would expect that the communication time should increase. While this applies to the bruck, ring, pair and the native algorithms, it is not the case for the simple algorithm: the communication time actually decreases as *MAX_IMBALANCE_FACTOR* increases when $MAX\_IMBALANCE\_FACTOR \leq 17$. The reason is that, in this cluster, 4 processors share the network interface card. With moderate imbalance in the process arrival pattern, different processors initiate their communications at different times, which reduces the resource contention and improves communication efficiency.

Part (b) of Figure 4.21 shows the performance when the message size is $64KB$. When $MAX\_IMBALANCE\_FACTOR \leq 9$, the pair algorithm is noticeably more efficient than the ring algorithm, which in turn is faster than the simple algorithm. However, the simple algorithm offers the best performance when $MAX\_IMBALANCE\_FACTOR \geq 33$. The ring

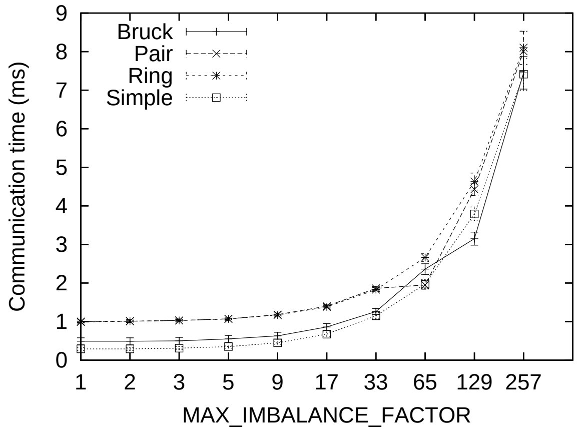


(a) 1 Byte

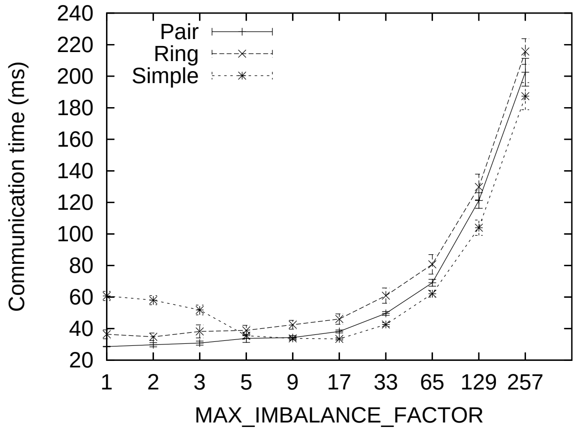


(b) 64KB

Figure 4.21: MPI_Alltoall on Lemieux (32 processors)



(a) 1 Byte



(b) 64KB

Figure 4.22: MPI_Alltoall on the Beowulf cluster

algorithm is also better than the pair algorithm when $MAX\_IMBALANCE\_FACTOR \geq 65$. For this message size, the native *MPI_Alltoall* routine performs worse than all three algorithms when $MAX\_IMBALANCE\_FACTOR \leq 65$. Figure 4.21 (b) also shows that each algorithm performs very differently under process arrival patterns with different imbalance factors.

Parts (a) and (b) of Figure 4.22 show the results on the Beowulf cluster for the two message sizes. The trend on the Beowulf cluster is similar to that on Lemieux. From this study, it can be seen that while algorithms behave differently under different message sizes and on different platforms, the common observation in all experiments is that each algorithm performs very differently under different process arrival patterns. Moreover, the algorithm that performs better under a balanced process arrival pattern may perform worse when the process arrival pattern becomes more imbalanced (e.g. the pair versus the simple algorithm). This indicates that evaluating collective communication algorithms based on the balanced arrival pattern only is insufficient for finding efficient practical algorithms. In order for MPI collective communication routines to achieve high performance, the impact of imbalanced process arrival patterns on communication algorithms must be taken into account.

4.2.5 Implication of the Imbalanced Process Arrival Pattern

It is unlikely that one algorithm for a collective operation can achieve high performance for all process arrival patterns. One potential solution is to have the communication library maintain a set of algorithms and to dynamically select the best algorithm. The work presented in the next section (STAR-MPI) is an attempt in this direction. Regardless of how the library is implemented, a common issue that must be addressed is how to select a good collective communication algorithm. Addressing this issue requires the understanding of the implication of the imbalanced process arrival pattern, which is discussed next.

4.2.5.1 Algorithms for MPI Collective Operations

Algorithms for MPI collective operations can be classified into three types, which are called *globally coordinated algorithms*, *store-and-forward algorithms*, and *directly communicating algorithms*. Globally coordinated algorithms require the coordination of all processors to carry out the operation efficiently. Such algorithms include all of the *phased algorithms*, where the complex collective operation is partitioned into phases and each process is assigned a certain duty in each phase. Examples of the globally coordinated algorithms include the ring and pair all-to-all algorithms [77], the bruck all-to-all algorithm [7], and the recursive doubling all-gather algorithm [77]. An imbalanced process arrival pattern may have a very large impact on this type of algorithms since the processes may not be able to coordinate as they are designed.

The store-and-forward algorithms use the store-and-forward mechanism to carry out the collective operations. Examples include various logical tree based broadcast algorithms and the logical ring all-gather algorithm [77]. In store-and-forward algorithms, there is some coordination among the processes and the imbalanced process arrival pattern may have some impact on such algorithms: when a process that stores and forwards a message in the operation arrives late, all processes that depend on the forwarded message will be affected.

The directly communicating algorithms do not require any global coordination among processes and do not use the store-and-forward mechanism. Such algorithms require the least amount of coordination among the three types of algorithms. Examples include the simple all-to-all algorithms [77] and the flat tree broadcast algorithm (the root sends to each of the receivers in sequence). An imbalanced process arrival pattern has the smallest impact on such algorithms.

The globally coordinated algorithms usually achieve the best performance among the three types of algorithms for the same operation when the process arrival pattern is balanced. As a result, such algorithms are commonly adopted in MPI implementations. However, the imbalanced process arrival pattern also affects this type of algorithms more than other types of algorithms. The study in the previous subsection suggests that globally coordinated algorithms may be over-rated while directly communicating algorithms may be under-rated. Hence, it would be meaningful to re-evaluate different types of MPI collective communication algorithms, taking the process arrival pattern characteristics into consideration.

4.2.5.2 MPI Collective Operations

There are two types of MPI collective operations: *globally synchronized* and *not globally synchronized* operations. The not globally synchronized operations include *MPI_Bcast*, *MPI_Scatter*, *MPI_Gather*, and *MPI_Reduce*. In such operations, a process may exit before all processes arrive. However, some algorithms for these types of operations may require all processes to arrive before any process can exit. One example is the pipelined broadcast algorithm [2]. These types of algorithms are called globally synchronized algorithms. Although using a globally synchronized algorithm to realize a not globally synchronized operation may achieve high performance under the assumption that the process arrival pattern is balanced, the study in the previous subsection indicates that it may be unwise to use a globally synchronized algorithm to realize a not globally synchronized operation in a general purpose MPI library since the process arrival pattern is generally imbalanced.

The globally synchronized operations include *MPI_Alltoall*, *MPI_Allreduce*, and *MPI_Allgather*. Such operations require either an explicit or implicit global synchronization: a process can exit such operations only after all processes arrive. For globally synchronized operations, the total communication time will be proportional to the worst case imbalanced factor (assuming the process arrival time is uniformly distributed between the first and the last process arrival time) when the worst case imbalanced factor is very large. For this type of operations, it does not make much sense to consider the cases when the imbalanced factor is very large (in this case, all algorithms will have similar performance). Hence, the library developers should focus on finding good algorithms for the cases when the imbalanced factor is not very large.

4.3 STAR-MPI

The study in the previous section indicated that the process arrival pattern of collective operations in MPI programs is likely to be imbalanced. Since this important aspect of application behavior can have a significant impact on the performance of a collective communication algorithm, it must be considered when developing efficient MPI collective communication routines: the impact of process arrival pattern must be included in the performance evaluation of a collective communication algorithm. This means that, in order for MPI collective communication routines to achieve high performance, they need to adapt not only to platform parameters but also to application behavior. To achieve this adaptability, the performance of communication algorithms must be measured in the context of the application on the platform.

In this section, another prototype DF library is presented: **S**elf **T**uned **A**daptive **R**outines for **MPI** collective operations (STAR-MPI) [22]. STAR-MPI is a library of collective communication routines that are capable of carrying out the tuning process in the context of application execution. Unlike STAGE-MPI, which generates collective routines that practically adapt only to the platform, STAR-MPI routines are adaptable to both platforms and applications. STAR-MPI maintains a set of algorithms for each operation and applies the Automatic Empirical Optimization of Software (AEOS) technique [82] at run time to dynamically select (tune) the algorithms as the application executes. STAR-MPI targets programs that invoke a collective routine a large number of times (programs that run for a large number of iterations).

One major issue in STAR-MPI is whether the AEOS technique can effectively select good algorithms at run time. Hence, the primary objective is to develop AEOS techniques that can find the efficient algorithms at run time. Under the condition that efficient algorithms can be found, the secondary objective is to reduce the tuning overhead. STAR-MPI incorporates various techniques for reducing the tuning overhead while selecting an efficient algorithm. The performance of STAR-MPI is evaluated and the results show that (1) STAR-MPI is robust and effective in finding efficient MPI collective routines; (2) the tuning overheads are manageable when the message size is reasonably large; and (3) STAR-MPI finds the efficient algorithms for the particular platform and application, which not only out-performs traditional MPI libraries to a large degree, but also offers better performance in many cases than STAGE-MPI that has a super-set of algorithms.

4.3.1 The STAR-MPI Library

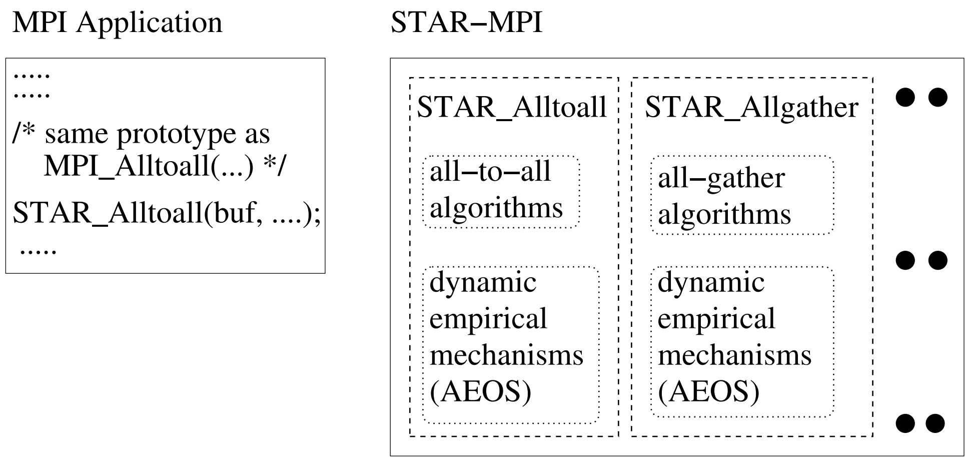

MPI Application
.....
.....
/* same prototype as MPI_Alltoall(...) */
STAR_Alltoall(buf, ...);
.....
STAR−MPI
STAR_Alltoall
all−to−all algorithms
dynamic empirical mechanisms (AEOS)
STAR_Allgather
all−gather algorithms
dynamic empirical mechanisms (AEOS)

Figure 4.23: High level view of STAR-MPI

The high level view of the STAR-MPI library is shown in Figure 4.23. STAR-MPI is an independent layer or library that contains a set of collective communication routines whose prototypes are the same as the corresponding MPI collective communication routines. MPI programs can be linked with STAR-MPI to accesses the adaptive routines. As shown in the figure, an *MPI_Alltoall* in an application is replaced with *STAR_Alltoall* routine. Different from traditional MPI libraries, each STAR-MPI routine has access to an algorithm repository that contains multiple implementations for the operation. In addition, each STAR-MPI routine incorporates a dynamic Automatic Empirical Optimization of Software (AEOS) [82] module, which performs self-monitoring and self-tuning during program execution. By maintaining multiple algorithms that can achieve high performance in different situations for each operation and using a dynamic empirical approach to select the most efficient algorithm, STAR-MPI is able to adapt to the application and platform.

STAR-MPI runs over MPICH. The routines supported in STAR-MPI include *MPI_Alltoall*, *MPI_Allgather*, *MPI_Allgatherv*, *MPI_Allreduce*, *MPI_Bcast*, and *MPI_Reduce*. STAR-MPI is designed for Ethernet switched clusters. All algorithms in STAR-MPI come from the algorithm repository of STAGE-MPI, which were designed for Ethernet switched clusters. Hence, STAR-MPI achieves the best results on Ethernet switched clusters, although it can also tune routines for other types of clusters. In the following, the algorithms maintained in STAR-MPI are summarized first and the dynamic AEOS technique is described second.

4.3.1.1 Collective Algorithms in STAR-MPI

As shown in Figure 4.23, each collective routine in STAR-MPI includes an algorithm repository that contains a set of communication algorithms. These algorithms can potentially achieve high performance in different situations. The organization of the algorithm repository is similar to that of STAGE-MPI, which is shown in Section 4.1.1.1. It includes both topology-unaware algorithms and topology-specific algorithms. The topology-specific algorithms are automatically generated based on the topology information when it is available. In cases when the topology information is not available, only the topology-unaware algorithms are used in the tuning. Meta-data is associated with each algorithm to describe important properties of the algorithms. One example of the meta-data is the range of the message sizes where the algorithm can be efficiently applied. The AEOS algorithm in STAR-MPI may use the meta-data information to select the communication algorithms that would be included in a particular tuning process.

Selecting algorithms to be included in STAR-MPI is very important for the performance since (1) the number of algorithms in the repository directly affects the time to tune the routine, and (2) the performance of each of the algorithms selected directly affects the program execution time since it must be executed even if it is not used in the final routine. Hence, the criteria for including an algorithm in STAR-MPI are more strict than those for including an algorithm in STAGE-MPI: the set of algorithms used in STAR-MPI is a subset of the algorithms in STAGE-MPI. The selection represents a trade-off between overhead and effectiveness. Including more algorithms makes STAR-MPI more effective and robust, but will introduce more overhead in the tuning process. The selection is based on previous experience with STAGE-MPI. First, algorithms that are rarely selected by STAGE-MPI are removed. Second, some algorithms in STAGE-MPI have a large parameter space. It takes a long tuning time to obtain the algorithm with the best parameter values. STAR-MPI replaces such an algorithm (with a large parameter space) by a small number of most promising algorithm instances.

Next, the STAR-MPI algorithms for *MPI_Alltoall*, *MPI_Allgather*, and *MPI_Allreduce* are mentioned. These routines are used in the performance evaluation. *MPI_Allgatherv* has exactly the same sets of algorithms as *MPI_Allgather*. Details about each of the algorithms can be found in the discussion of STAGE-MPI repository in Section 4.1.1.1.

Algorithms for MPI_Alltoall

There are 13 all-to-all algorithms in STAR-MPI: *simple, 2D mesh, 3D mesh, recursive doubling (rdb), bruck, ring, ring with light barrier, ring with MPI barrier, pair, pair with light barrier, pair with MPI barrier, topology-specific with sender-sync,* and *topology-specific with receiver-sync.* The *2D mesh*, *3D mesh*, *rdb*, and *bruck* algorithms are designed for small messages. STAR-MPI only uses them to tune for messages up to 256 bytes.

Algorithms for MPI_Allgather

STAR-MPI maintains 12 algorithms for *MPI_Allgather*. The all-gather communication pattern is a special all–to–all communication pattern (sending the same copy of data to each node instead of sending different messages to different nodes). The STAR-MPI algorithm repository for *MPI_Allgather* includes the following algorithms that work similar to their all-to-all counterparts: *simple*, *2D mesh*, *3D mesh*, *rdb*, *ring*, *ring with light barrier*, *ring with*

MPI barrier, *pair*, *pair with light barrier*, *pair with MPI barrier*. STAR-MPI also includes the *bruck* all-gather algorithm [7], which is different from the bruck all-to-all algorithm. Details can be found in [7, 77]. In addition, STAR-MPI repository includes the topology-specific logical ring (TSLR), which is described in Section 3.3. This algorithm is included when topology information is available. Otherwise, the topology-unaware logical ring (LR) is included instead.

Algorithms for MPI_Allreduce

STAR-MPI maintains 20 algorithms for *MPI_Allreduce*. These algorithms can be classified into three types.

In the first-type algorithms, the *MPI_Allreduce* operation is performed by first using an *MPI_Allgather* to gather the data in all nodes and then performing the reduction operation. The all-gather has the following variations: *bruck, 2D mesh, 3D mesh, rdb,* and *ring.*

The second-type algorithms are variations of the Rabenseifner algorithm [64], where the all-reduce operation is performed by a reduce-scatter operation followed by an all-gather operation. The reduce-scatter is realized by recursive halving [64] and the all-gather implementation has the following variations: *simple*, *2D mesh*, *3D mesh*, *rdb*, *bruck*, *ring*, *ring with light barrier*, *ring with MPI barrier*, and *TSLR/LR*. The term *Rab1-x* denotes this type of algorithms with x all-gather implementations. For example, *Rab1-2D* means the variation with the *2D mesh* all-gather algorithm.

The third-type algorithms are also variations of the Rabenseifner algorithm [64], where the all-reduce operation is performed by a reduce-scatter operation followed by an all-gather operation. In this case, the reduce-scatter operation is realized by an all-to-all operation. The algorithm is denoted by the pair (*all-to-all*, *all-gather*). STAR-MPI maintains the following algorithms: (*ring, TSLR/LR*), (*ring with light barrier, TSLR/LR*), (*ring with MPI barrier, TSLR/LR*), (*ring, ring*), (*ring with light barrier, right with light barrier*), and (*ring with MPI barrier, ring with MPI barrier*). The notion *Rab2-(x, y)* will be used to denote this type of algorithms with the *(x, y)* algorithms. For example, *Rab2-(ring, ring)* denotes the *(ring, ring)* variation of this type of algorithms.

4.3.1.2 Dynamic AEOS Algorithm in STAR-MPI

Given a set of algorithms, the primary objective of the dynamic AEOS algorithm is to find the most efficient algorithm (among the set of algorithms) for an application running on a given platform. The second objective is to minimize the overheads. Next, the AEOS algorithm that achieves these two objectives is described.

Different MPI call sites or even the same call site invoked with different message sizes constitute different program contexts. As shown in the previous study of Section 4.2, MPI routines used in different program contexts usually have different program behavior. To achieve the maximum tuning effectiveness, routines in different contexts must be tuned independently. STAR-MPI addresses this issue as follows. For each MPI collective routine, STAR-MPI supports N independent but identical routines, where N is a parameter. Different call sites of the same MPI routine in an MPI program can be tuned independently. To deal with the case of invoking the same call site with different message sizes, STAR-MPI allows each call site to tune for a pre-defined number, X, of message sizes. If a call site has more than X different message sizes during the program execution, STAR-MPI tunes for the first X sizes and uses the default MPI routine for the rest of sizes. Note that in practice, a call site in an MPI program usually results in only a small number of message sizes, most call sites only have one message size. This arrangement allows the dynamic AEOS algorithm to focus on tuning for one message size on one call site to maximize the tuning effectiveness. In the rest of the section, it will be assumed that the AEOS algorithm is applied to tune one message size on one call site.

In the course of program execution, a STAR-MPI routine (for one message size in each call site) goes through two stages: *Measure_Select* and *Monitor_Adapt*. In the *Measure_Select* stage, in each invocation of the routine, one of the algorithms in the repository is used to realize the operation and the performance of the algorithm is measured. During the *Measure_Select* stage, all algorithms in the repository will be executed and measured a number of times. The number of times that each algorithm is executed and measured in this stage is a system parameter. At the end of the *Measure_Select* stage (all algorithms are executed and measured), an all-reduce operation is performed to compute the performance results on all processors and an algorithm is selected as the best algorithm based on the measured performance. The performance of all other algorithms is stored for future use.

Notice that for the whole *Measure_Select* stage, only one additional all-reduce communication (with a reasonable small message size) is performed. Note also that in this stage, less efficient algorithms end up being used to carry out the operation since their performance must be measured. After the *Measure_Select* stage, it is expected that the selected algorithm will deliver high performance for the subsequent invocations. However, this may not always occur due to various reasons. For example, the initial measurement may not be sufficiently accurate, or the workload in the application may change. To handle such situations, in the *Monitor_Adapt* stage, STAR-MPI continues monitoring the performance of the selected algorithm and adapts (changes the algorithm) when the performance of the selected algorithm deteriorates.

Figure 4.24 shows the details of the dynamic AEOS algorithm. In the figure, the use of *STAR_Alltoall* illustrates how to tune *MPI_Alltoall*. The AEOS algorithm is the same for all operations supported in STAR-MPI. It is important to understand that all internal states of *STAR_Alltoall* (or any other STAR-MPI collective routine) are static since it must be retained between invocations. Each time *STAR_Alltoall* is called, the algorithm first computes (line 1) the message size, x, for the operation. Once the message size is known, the algorithm can be in either of the two previously described stages depending on the value of *best_algorithm*$_x$. As shown in lines 3-6, if *best_algorithm*$_x$ points to an invalid communication algorithm index, denoted by NIL, then the algorithm is in the *Measure_Select* stage and calls the `Measure_Select()` routine. Otherwise, it is in the *Monitor_Adapt* stage and calls the `Monitor_Adapt()` routine.

The logic of the `Measure_Select()` routine (lines 7-18) is straight-forward. It runs and measures each algorithm ITER times. ITER is a parameter that can be controlled and is by default set to 10 in the current system. This number was determined experimentally; it is a trade-off between the tuning overhead and measurement accuracy. When all communication algorithms are examined, the `Dist_Time()` routine is called (line 17) to compute the communication time for all algorithms and distribute the results to all processors, and the `Sort_Alg()` routine is called (line 18) to sort the algorithms based on their performance and select the best algorithm (set the value for *best-algorithm*$_x$). Notice that the algorithm is selected based on the *best* performance measured.

Once *best_algorithm*$_x$ is set, the AEOS algorithm enters the *Monitor_Adapt* stage. In this stage, the algorithm pointed by *best-algorithm*$_x$ is used to realize the operation. The AEOS

```
ITER: number of iteration to examine an algorithm
δ: monitoring factor, initialized to 2
T: threshold used to switch between algorithms
TOTAL_ALGS: total number of algorithms to examine
func: pointer to a given function pointed by $index_x$

$best\_algorithm_x \leftarrow NIL$;
STAR_Alltoall(sbuf, scount, stype, ...)
1    MPI_Type_size(stype, & size)
2    $x \leftarrow$ scount * size
3    if ($best\_algorithm_x == NIL$)
4        Measure_Select(sbuf, scount, stype..., x)
5    else
6        Monitor_Adapt(sbuf, scount, stype, ..., x)

$index_x \leftarrow iter_x \leftarrow 0$
Measure_Select(sbuf, scount, stype, ..., x)
7    func ← Alltoall_Alg($index_x$)
8    $t_0$ ← MPI_Wtime()
9    func(sbuf, scount, ...)
10   $t_1$ ← MPI_Wtime()
11   le_time[$index_x$][$iter_x$] ← $t_1$ - $t_0$
12   $iter_x$++
13   if ($iter_x$ == ITER)
14       $iter_x \leftarrow 0$
15       $index_x$++
16   if ($index_x$ == TOTAL_ALGS)
17       Dist_Time(le_time, ge_time, best_time)
18       best-algorithm$_x$ ← Sort_Alg(best_time, x)

Monitor_Adapt(sbuf, scount, stype, ..., x)
19   func ← Alltoall_Alg($best\_algorithm_x$)
20   $t_0$ ← MPI_Wtime()
21   func(sbuf, scount, ...)
22   $t_1$ ← MPI_Wtime()
23   total[0] ← total[0] + ($t_1$ - $t_0$)
24   if ($\delta$*ITER - $monitor_x \leq$ ITER)
25       total[1] ← total[1] + ($t_1$ - $t_0$)
26   $monitor_x$++
27   if ($monitor_x$ == $\delta$ * ITER)
28       MPI_Allreduce(total, ave, 2, .., MPI_SUM, ..)
29       ave[0] ← total[0] / $monitor_x$
30       ave[1] ← total[1] / ITER
31       if (ave[0] < (1+$\epsilon$) * best_time[1])
32           $\delta \leftarrow \delta$ * 2
33       else if (ave[0] $\geq$ (1+$\epsilon$) * best_time[1])
34           if (ave[1] $\geq$ (1+$\epsilon$) * best_time[1])
35               best_time[$best\_algorithm_x$] ← ave[0]
36               best-algorithm$_x$ ← Sort_Alg(best_time, x)
37               $\delta \leftarrow 2$
38       $monitor_x$ ← total[0] ← total[1] ← 0

Dist_Time(le_time, ge_time, best_time)
39   MPI_Allreduce(le_time, ge_time,..MPI_SUM,..)
40   foreach $i$ in 0 .. TOTAL_ALG
41       foreach $j$ in 0 .. ITER
42           ge_time[i][j] ← ge_time[i][j] / nprocs
43   foreach $i$ in 0 .. TOTAL_ALG
44       best_time[$i$]←$\min_{0 \leq j < ITER}$(ge_time[$i$][$j$])
```

Figure 4.24: Using STAR-MPI algorithm to tune MPI_Alltoall

task in this stage is to monitor the performance of the selected communication algorithm and to change (adapt) to another algorithm when the performance of the selected algorithm deteriorates.

The `Monitor_Adapt()` routine is shown in lines 19-38. The logic is as follows. First, the algorithm pointed by *best-algorithm$_x$* is used to realize the operation and the performance on each processor is measured. The monitoring is done locally (no global communication) during the monitoring period, which is defined as $\delta * ITER$ invocations, where δ is a variable whose value is initialized to be 2. At the end of the monitoring period, an all-reduce operation is performed to compute the performance of the selected algorithm and distribute the performance results to all processors. If the average communication time of the selected algorithm during the monitoring period is less than $(1 + \epsilon) *$ *second best time*, the length of the monitoring period is doubled. Here, ϵ is a system parameter, currently set to 10%. If the average communication is more than $(1 + \epsilon) *$ *second best time*, there are two cases. If the average communication time of the last ITER invocations is also larger than $(1 + \epsilon) *$ *second best time*, this indicates that the selected algorithm may not be as efficient as the second best algorithm and, thus, the second best algorithm is now selected. The average time of the replaced algorithm is recorded and algorithms are re-sorted based on their performance. When a new algorithm is selected, δ is reset to 2. If the average communication time of the last ITER invocations is less than $(1+\epsilon) *$ *second best time*, the bad performance measured may be caused by some special events and the AEOS algorithm resets $\delta = 2$ so that the selected algorithm can be monitored more closely.

The monitoring is critical to ensure that STAR-MPI will eventually find an efficient algorithm. A number of trade-offs between monitoring overheads and algorithm effectiveness are made in the Monitor_Adapt routine. First, the length of the monitoring period, which is controlled by δ, doubles every time the selected algorithm continues to perform well. This reduces the monitoring overhead: if the selected algorithm continues to perform well, the total number of all-reduce operations in the *Monitor_Adapt* stage is a logarithm function of the total number of invocations. However, this creates a chance for STAR-MPI to adapt too slowly due to large monitoring periods. In practice, an upper bound can be set for δ to alleviate this problem. Second, a simple heuristic is used to decide whether the selected algorithm is still performing well. A more complex statistical approach may improve the monitoring accuracy by better filtering out noise in the measurement or program execution. Such an approach will incur more computation and more communication in the *Monitor_Adapt* stage. The simple approach is adopted since it works quite well in the experiments.

4.3.1.3 Enhancing Measure_Select by Algorithm Grouping

As can be seen from the previous discussion, most of the tuning overheads occur in the *Measure_Select* stage. When the message size is reasonably large, the bookkeeping overhead in STAR-MPI is relatively small. However, the penalty for using less efficient algorithms to realize an operation can be potentially very high. Two parameters affect this penalty:

the parameter ITER and the number of less efficient algorithms in the repository. Hence, ideally, one would like to reduce the number of algorithms as much as possible. However, the problem is that reducing the number of algorithms may make the system less robust and that before the algorithm is executed and measured, it is difficult to decide which algorithm is more efficient than other algorithms.

Algorithm grouping is one way to reduce the number of algorithms to be probed without sacrificing the tuning effectiveness. Algorithm grouping is based on the observation that a collective communication algorithm usually optimizes for one or multiple system parameters. When a system parameter has a strong impact on the performance of a collective operation, the algorithms that optimize this parameter tend to out-perform other algorithms that do not consider this parameter. Based on this observation, algorithm grouping groups algorithms based on their optimization objectives. For example, the *2D mesh*, *3D mesh*, *rdb*, and *bruck* algorithms for *MPI_Alltoall* all try to reduce the startup overhead in the operation by reducing the number of messages. If the startup overhead in an operation is important, any of these algorithms will out-perform other algorithms that do not reduce the number of messages. Hence, these four algorithms can be joined into one group. Once all algorithms are joined into groups, the `Measure_Select()` routine can first identify the best performing groups by comparing algorithms in different groups (one algorithm from each group) and then determine the best performing algorithm by evaluating all algorithms in that group. This two-level tuning scheme reduces the number of algorithms to be measured in the *Measure_Select* phase while maintaining the tuning effectiveness (theoretically, all algorithms are still being considered). Notice that algorithm grouping also affects the *Monitor_Adapt* stage: when a new algorithm in a new group is selected, if the algorithms in the new group have not been probed, the system must first examine all algorithms in the group before selecting the best performing algorithm. In the remainder of the section, the AEOS algorithm without grouping is called the *basic* AEOS algorithm and with grouping the *enhanced* AEOS algorithm.

The effectiveness of algorithm grouping depends on how the algorithms are grouped. In STAR-MPI, the algorithms are grouped based on the previously described performance model for STAGE-MPI in Section 4.1.1.1. Algorithms that optimize the same set of parameters are merged in one group. Specifically, the 13 all-to-all algorithms are partitioned into 6 groups: group 1 contains *simple*; group 2 (used only for small messages ($\leq 256B$))

contains *rdb*, *2D mesh*, *3D mesh*, and *bruck*; group 3 contains *ring* and *pair*; group 4 contains *ring with light barrier* and *pair with light barrier*; group 5 contains *ring with MPI barrier* and *pair with MPI barrier*; group 6 contains the two topology specific algorithms. The 12 algorithms for *MPI_Allgather* are partitioned into 6 groups: group 1 contains *simple*; group 2 contains *rdb*, *2D mesh*, *3D mesh*, and *bruck*; group 3 contains *ring* and *pair*; group 4 contains *ring with light barrier* and *pair with light barrier*; group 5 contains *ring with MPI barrier* and *pair with MPI barrier*; group 6 contains either the topology-unaware logical ring (LR) or the topology specific logical ring (TSLR) algorithm (if topology information is known). The 20 *all-reduce* algorithms are partitioned into 3 groups based on the three types of algorithms. Notice that although this grouping scheme may not be optimal, it allows for evaluating the grouping technique that is proposed to improve dynamic AEOS scheme.

In general, grouping trades tuning overheads with the quality of the selected algorithm: the best performing algorithm may not be selected with grouping. However, in all tests, the enhanced STAR-MPI selected as good (or virtually as good as) an algorithm as did the basic AEOS algorithm while significantly reducing the overhead.

4.3.2 Performance Evaluation

Most of the experiments are performed on Ethernet-switched clusters since STAR-MPI is equipped with algorithms that are designed for Ethernet-switched clusters. To demonstrate the robustness of the STAR-MPI technique, STAR-MPI is also tested on the Lemeiux machine at Pittsburgh Supercomputing Center (PSC) [60]. The nodes of the Ethernet clusters are Dell Dimension 2400 with a 2.8GHz P4 processor, 128MB of memory, and 40GB of disk space. All machines run Linux (Fedora) with 2.6.5-1.358 kernel. The Ethernet card in each machine is Broadcom BCM 5705 with the driver from Broadcom. These machines are connected to Dell PowerConnect 2224 and Dell PowerConnect 2324 100Mbps Ethernet switches. The topologies used in the experiments are shown in Figure 4.25. Part (a) of the figure is a 16-node cluster connected by a single switch while part (b) shows a 32-node cluster connected by 4 switches, with 8 nodes attached to each switch. The topologies are referred to as topology (a) and topology (b), respectively.

As discussed earlier, there are two major performance issues in STAR-MPI. First, for STAR-MPI to be efficient, the AEOS technique must be able to select good communication algorithms at run-time. To examine the capability of STAR-MPI in selecting good commu-

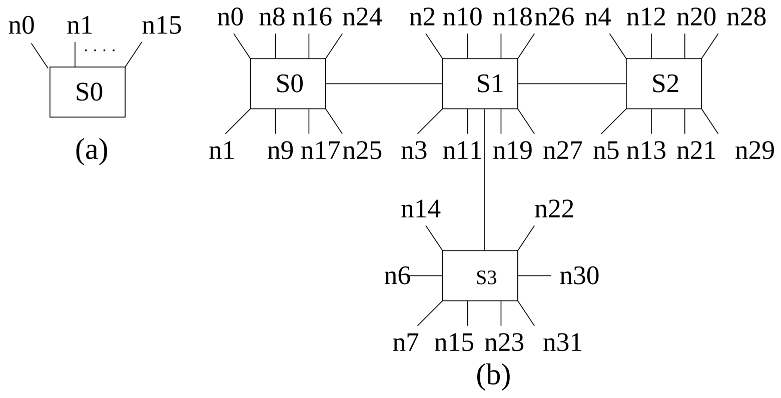


Figure 4.25: Topologies used in the experiments

nication algorithms, the performance of STAR-MPI (STAR) is compared with the original MPICH 2.1.0.1 (MPICH) and STAGE-MPI (STAGE). Second, STAR-MPI introduces overhead in both the `Measure_Select` and `Monitor_Adapt` stages. In the `Measure_Select` stage, less efficient communication algorithms are executed to carry out the operations. Such overhead is denoted as O^c_{MS}. Additional overhead is introduced in this stage to execute the AEOS logic (e.g. measuring, computing, and recording the performance of all communication algorithms). The term O^a_{MS} denotes such overhead. In the `Monitor_Adapt` stage, the overhead is introduced to monitor the performance and to execute the logic to determine whether the current communication algorithm should be changed. The overhead in the `Monitor_Adapt` stage is denoted as O_{MA}. This subsection is organized as follows. First, the basic AEOS algorithm is compared with the enhanced AEOS algorithm that uses algorithm grouping. The capability of STAR-MPI in selecting good communication algorithms is then studied in a number of application programs. After that, the overhead of STAR-MPI is evaluated. Finally, the results of experiments on the Lemieux cluster at PSC are presented.

4.3.2.1 Basic AEOS .vs. Enhanced AEOS

For STAR-MPI to be efficient, it must (1) be able to find the efficient communication algorithms and (2) find the algorithms as quickly as possible. The communication algorithm found at the end of the *Measure_Select* stage usually (but not always) offers reasonably good performance and is used thereafter. Hence, a good indication of the performance of an AEOS algorithm is (1) the quality of the communication algorithm selected at the end of *Measure_Select*, and (2) the duration of the *Measure_Select* stage, which measures how fast the AEOS algorithm can find the selected algorithm. In the following, the basic and enhanced AEOS algorithms are compared using these two metrics.

```
for (i = 0; i < 500; i++) {
    ... // computation that lasts roughly 5 times
        // the collective operation time
    start = MPI_Wtime();
    MPI_Alltoall(...);
    elapsed += (MPI_Wtime() - start);
}
```

Figure 4.26: An example micro-benchmark

Micro-benchmarks that are similar to the one shown in Figure 4.26 are used in the comparison. This micro-benchmark simulates programs with a perfect computation load distribution. The main loop contains both computation and collective communication. The time for the computation in the loop is set to be roughly 5 times the total communication time. The elapsed time for the communication is measured and reported.

Table 4.13 shows the number of invocations in the *Measure_Select* stage, the total communication time in this stage, and the algorithms selected by the basic and enhanced AEOS algorithms. As expected, the enhanced scheme greatly reduces the number of invocations and the time in the *Measure_Select* stage. Moreover, the algorithms selected by the two schemes are mostly the same. In cases when the selected algorithms are different (e.g. $128KB$ all-to-all and $128KB$ all-reduce), the performance of the different communication algorithms is very similar. Experiments with different message sizes and different topologies are conducted and similar observations are obtained. Hence, it can be concluded that the enhanced scheme is more efficient than the basic scheme. In the rest of the subsection, only the results of STAR-MPI with the enhanced AEOS algorithm are reported.

4.3.2.2 Application Results

Since STAR-MPI targets programs that invoke collective routines a large number of times, the application benchmarks are selected such that they (1) run for a large number of iterations

Table 4.13: Basic AEOS vs. enhanced AEOS on topology (a)

operation	msg size	Measure_Select	basic	enhanced
all-to-all	2KB	# of invocations	90	50
		time (ms)	2181	1183
		algorithm	*simple*	*simple*
	128KB	# of invocations	90	60
		time (ms)	31953	19018
		algorithm	*ring light barrier*	*pair light barrier*
all-gather	2KB	# of invocations	120	60
		time (ms)	2055	1229
		algorithm	*simple*	*simple*
	128KB	# of invocations	120	60
		time (ms)	41233	18335
		algorithm	*TSLR*	*TSLR*
all-reduce	2KB	# of invocations	200	110
		time (ms)	2413	1571
		algorithm	*Rab1-2D*	*Rab1-2D*
	128KB	# of invocations	200	110
		time (ms)	25319	7895
		algorithm	*Rab1-3D*	*Rab1-bruck*

and (2) have significant collective communications. To achieve high performance for this type of programs, it is critical that STAR-MPI must eventually select efficient communication algorithms to carry out the collective operations. The results shown here mainly reflect the capability of STAR-MPI in selecting efficient communication algorithms.

Four applications are used in the evaluation: FFTW [27], LAMMPS [45], NTUBE [66], and NBODY [57]. FFTW [27] is a C library of routines for computing the discrete Fourier transform in one or more dimensions, of arbitrary input size, and of both real and complex data. When using the benchmark test driver, the value of l (linear size) is 1500 and the value of *nfft* (number of Fourier transforms to execute) is 500. The LAMMPS (Large-scale

Atomic/Molecular Massively Parallel Simulator) [45] benchmark models the assembly of particles in a liquid, solid, or gaseous state. In the experiments, the program ran with 1720 copper atoms for 10000 iterations. The NTUBE (Nanotube) program performs molecular dynamics calculations of thermal properties of diamond [66]. In the evaluation, NTUBE runs for 1000 steps and simulate 25600 atoms. Finally, NBODY [57] simulates over time steps the interaction, in terms of movements, position and other attributes, among the bodies as a result of the net gravitational forces exerted on one another. The code ran for 1000 steps with 8000 bodies on topology (a) and 25600 bodies on topology (b). Note that the number of iterations or time steps for each benchmark is chosen such that it is sufficient enough (1) to allow STAR-MPI routines finish the *Measure_Select* stage and (2) to achieve considerable performance gains that will amortize the overheads associated with the STAR-MPI technique. For the different benchmarks, Table 4.14 shows the major MPI collective routines and the message sizes for topologies (a) and (b). These routines account for a significant portion of the total application times and are tuned using STAR-MPI.

Table 4.14: Collective routines in the applications on different topologies

program	routine	topo.	msg size
FFTW	MPI_Alltoall	(a)	141376B
		(b)	33856B
LAMMPS	MPI_Allreduce	(a)	42382B
		(b)	42382B
NTUBE	MPI_Allgatherv	(a)	256000B
		(b)	128000B
NBODY	MPI_Allgather	(a)	20000B
		(b)	32000B

Next, Table 4.15 shows the different communication algorithms selected in STAR, STAGE, and MPICH to realize the collective operations in the four application benchmarks. The STAR algorithms in the table are the final algorithms selected for the application. Some algorithms in the table are not included in STAR-MPI and thus have not been discussed. The description of these algorithms can be found in Section 4.1.1.1. There are two main observations. First, MPICH has limited software adaptability as it only considers the message size and the number of processors. In particular, for all benchmarks except NBODY, the communication algorithms that MPICH uses are the same across both topologies. In the case of NBODY, the message sizes of the all-gather operation on the two topologies were not in the same range, which caused MPICH to use two different communication algorithms. Since MPICH does not take into consideration application behavior and all aspects of the platform, its predetermined selection of algorithms will be inadequate in many cases. Second, with the exception of NTUBE and NBODY on topology (b), the table shows that the STAR-MPI versions of communication algorithms for the different collective operations are quite different than (and superior to) the ones used by STAGE-MPI. The reason these algorithms are picked by STAR-MPI but not STAGE-MPI, although the algorithms are available in its repository, is that STAGE-MPI has only architectural information and selects the best algorithms for Mpptest, not the applications. As a result, STAGE-MPI may not yield the most efficient algorithm for an application since the program context is unavailable. STAR-MPI attains full adaptability for the collective operations because it has access to application information.

Table 4.15: Communication algorithms used in STAR, STAGE, and MPICH on different topologies (T)

benchmark	T.	STAR	STAGE	MPICH
FFTW	(a)	*pair with light barrier*	*pair with MPI barrier*	*pair*
(*MPI_Alltoall*)	(b)	*pair with light barrier*	*tuned pair with N MPI barrier*	*pair*
LAMMPS	(a)	*Rab1-ring with light barrier*	*Rab1-tuned*	*MPICH Rab*
(*MPI_Allreduce*)	(b)	*Rab1-rdb*	*Rab2-(tuned, tuned)*	*MPICH Rab*
NTUBE	(a)	*TSLR*	*pair with MPI barrier*	*LR*
(*MPI_Allgatherv*)	(b)	*TSLR*	*TSLR*	*LR*
NBODY	(a)	*simple*	*TSLR*	*rdb*
(*MPI_Allgather*)	(b)	*TSLR*	*TSLR*	*LR*

The results for the application benchmarks for MPICH, STAR, and STAGE are summarized in Table 4.16. Note that in all cases, STAR overhead is included in the presented results, while MPICH and STAGE both have no run-time overhead. First, the ability of STAR to select better communication algorithms than MPICH and STAGE is evident in the significant performance gains shown in the table. For all benchmarks running on the two topologies, except for NTUBE on topology (a), STAR is superior to MPICH. For example, for the FFTW benchmark, STAR achieves a 64.9% and 31.7% speedups over MPICH on topology (a) and topology (b), respectively. Also, substantial gains are seen for LAMMPS

(85.6%) and NTUBE (408.9%) on topology (b). The result for the NTUBE benchmark on topology (a) shows that STAR performs slightly worse than MPICH. This is because both STAR and MPICH use the same communication algorithm to realize the all-gatherv operation in the benchmark, with STAR paying the extra tuning overhead.

Comparing STAR to STAGE shows that in many cases STAR is also superior. For example, on FFTW, STAR speedup relative to MPICH is much larger than that of STAGE(65% to 11% on topology (a) and 32% to 7.7% on topology (b)). STAR speedup is also much greater on LAMMPS (b) and NBODY (a), and STAR does not slow down on NTUBE (a), as described earlier, whereas STAGE does. This demonstrates the effectiveness of STAR that has a subset of algorithms in selecting better communication algorithms than STAGE, which has a super-set of algorithms. In two of the other cases (LAMMPS (a), NBODY (b)), the performance is similar, with STAGE slightly better. The one exception is on NTUBE (b), where STAGE speedup is much larger than STAR. Let us look at these last three cases next.

Table 4.16: Application completion times on different topologies

benchmark	topo.	STAR	STAGE	MPICH
FFTW	(a)	350.8s	519.6s	578.6s
	(b)	636.3s	778.0s	838.1s
LAMMPS	(a)	9780s	9515s	11040s
	(b)	1991s	2432s	3696s
NTUBE	(a)	568.0s	725.0s	566.0s
	(b)	758.4s	601.0s	3860s
NBODY	(a)	4002s	4268s	4304s
	(b)	2167s	2120s	2946s

Table 4.17 shows the performance of STAR with and without overhead, relative to STAGE. The performance of STAR without overhead, denoted as STAR', is obtained by running the final routine selected by STAR-MPI without the tuning and monitoring overheads. From Figure 4.17, it can be seen that STAR-MPI without overheads performs at least as good as STAGE, which indicates that the performance penalty (versus STAGE) is due to the overheads. As will be shown soon, the overhead is mainly introduced in the *Measure_Select* stage. The AEOS algorithm in STAR-MPI is robust. If applications run

for more iterations, the tuning overheads will be amortized. For example, if NTUBE runs for 2000 instead of 1000 time steps, the absolute STAR overhead would remain roughly the same, while the relative STAR overhead would decrease substantially. Notice that STAGE-MPI also has overhead: STAGE-MPI must be run over a significant period of time when a new platform is encountered. STAGE-MPI overheads are not considered in this experiment. Note also that the NTUBE (a) result shows that tuning with Mpptest can sometimes lead to the algorithms that significantly degrade the performance for an application. This is a major limitation of STAGE-MPI.

Table 4.17: Application completion times

benchmark	topo.	STAR	STAGE	STAR'
LAMMPS	(a)	9780s	9515s	9420s
NTUBE	(b)	758.4s	601.0s	601.0s
NBODY	(b)	2167s	2120s	2120s

4.3.2.3 STAR-MPI Overhead

Using the micro-benchmarks similar to the code in Figure 4.26, the overhead of STAR-MPI is examined in depth. The measured overheads include the overhead introduced by the execution of less efficient algorithms, O^c_{MS}, and the overheads for running the AEOS algorithm in both stages of STAR-MPI, namely O^a_{MS} and O_{MA}. Note that besides the O_{MA} overhead in the *Monitor_Adapt* stage, STAR-MPI may introduce extra overheads in this stage if it adapts to a different algorithm. While such adaptation is occasionally observed (all such adaptation occurs in the first monitoring period in the experiments), it is a low probability random event. Hence, only O^a_{MS}, O^c_{MS}, and O_{MA} are evaluated. In the following, the per invocation time of STAR-MPI collective routines in the *Measure_Select* and *Monitor_Adapt* stages is examined, and then the time is broken down in terms of the different overheads.

The per invocation times in the *Measure_Select* and *Monitor_Adapt* stages of STAR-MPI all-to-all, all-gather, all-reduce routines with different message sizes on topology (a) and (b) are shown in Table 4.18. The results are obtained using the micro-benchmark with 500 iterations, which include the iterations for both the *Measure_Select* and *Monitor_Adapt* stages. For example, for all-gather on topology (b) with message size $64KB$, the *Measure_Select*

Table 4.18: Per invocation time (ms) for collective operations in the micro-benchmark

operation	topo.	msg size	MPICH	STAR	
				Measure_Select	Monitor_Adapt
all-to-all	(a)	16KB	32.0	46.4	27.0
		64KB	305.7	192.1	114.6
		256KB	519.4	658.7	498.6
	(b)	16KB	324.0	366.4	323.2
		64KB	1493	1366	1108
		256KB	6079	7154	5716
all-gather	(a)	16KB	31.8	46.0	25.5
		64KB	111.6	147.2	104.5
		256KB	446.0	596.8	416.9
	(b)	16KB	87.8	293.8	58.66
		64KB	1532	1232	542
		256KB	6432	5037	2004
all-reduce	(a)	16KB	5.9	12.0	4.6
		64KB	18.7	24.4	18.6
		256KB	68.3	95.5	68.4
	(b)	16KB	18.4	26.4	9.68
		64KB	82.0	76.8	34.3
		256KB	335.4	250	128.6

stage occupies 60 invocations and the *Monitor_Adapt* stage occupies 440 invocations. For reference, the per invocation time for MPICH is also shown. There are a number of common observations for all operations on both topologies. First, the per invocation times are very different for the *Measure_Select* stage and the *Monitor_Adapt* stage. This is because the best performing algorithm significantly out-performs some of the algorithms in the repository. Second, as shown in the table, although the per invocation time of STAR-MPI in the *Measure_Select* stage reflects a quite significant overhead, such overhead is amortized (and then offset) by the gains due to the selection of a better communication algorithm during the post tuning or *Monitor_Adapt* stage. Third, in some cases (e.g. all-gather on topology (b) with message sizes of $64KB$ and $256KB$), STAR-MPI out-performs MPICH even in the *Measure_Select* stage. This is because some of the communication algorithms that STAR utilizes during tuning are more efficient than those in MPICH.

Table 4.19 breaks down the per invocation time in terms of the O^a_{MS}, O^c_{MS}, and O_{MA}

Table 4.19: Per invocation overheads (in millisecond) for collective operations in the micro-benchmark

operation	topo.	msg size	O^a_{MS}	O^c_{MS}	O_{MA}
all-to-all	(a)	16KB	0.04	46.4	0.01
		64KB	0.35	191.8	0.06
		256KB	1.60	657.1	0.30
	(b)	16KB	0.01	366.4	0.4
		64KB	0.01	1366.0	1.4
		256KB	0.01	7153.9	5.8
all-gather	(a)	16KB	0.01	45.9	0.03
		64KB	0.01	147.2	0.1
		256KB	0.7	596.1	0.2
	(b)	16KB	0.01	293.8	0.08
		64KB	0.01	1232	0.8
		256KB	0.01	5037.0	0.6
all-reduce	(a)	16KB	0.02	12.0	0.02
		64KB	0.01	24.39	0.03
		256KB	0.01	95.4	0.05
	(b)	16KB	0.05	26.3	0.03
		64KB	0.15	76.5	0.07
		256KB	0.5	249.5	0.20

overheads for the same STAR-MPI collective routines. For the different message sizes, the table shows that O^a_{MS} and O_{MA} are very small and account for less than 0.3% of the per invocation times (shown previously in Table 4.18) for the *Measure_Select* stage or the *Monitor_Adapt* stage. On the other hand, it is easily observed that O^c_{MS} can be very large. Thus, most of the overhead of STAR-MPI is due to the communication overhead in the tuning phase. This indicates that the selection of the set of communication algorithms is very critical for STAR-MPI to achieve high performance. Moreover, for different topologies, the table shows that STAR-MPI may introduce very different overheads. For example, the STAR-MPI all-gather routine introduces much more overhead on topology (b) than that on topology (a). This is because the topology can significantly affect the performance of a collective communication algorithm. Since the impact of topology is so significant, it may be worthwhile to develop a performance model that can take network topology into account and use the prediction from such a model to reduce the number of algorithms to be probed.

4.3.2.4 STAR-MPI on Lemieux

To further study the effectiveness and impact of STAR-MPI technique on different platforms, experiments are performed on the Lemieux supercomputing cluster, located in Pittsburgh Supercomputing Center (PSC)[60]. The machine consists of 750 Compaq Alphaserver ES45 nodes, each of which includes four 1-GHz SMP processors with 4GB of memory. The nodes are connected with a Quadrics interconnection network, and they run Tru64 Unix operating system. The experiments are conducted with a batch partition of 128 processors running on 32 dedicated nodes, although other jobs were concurrently using the network. The benchmarks are compiled with the native *mpicc* on the system and linked with the native MPI and ELAN libraries. ELAN is a low-level internode communication library that efficiently realizes many features of the Quadrics interconnection such as multicast.

The algorithms used in the collective communication routines in the native MPI library could not be identified. The Quadrics interconnect in this machine has very efficient hardware support for multicast. As a result, for collective operations that have a multicast or broadcast component, including all-gather, all-gatherv, and all-reduce, the native routines out-perform STAR-MPI (sometimes to a large degree) since all STAR-MPI algorithms are based on point-to-point primitives. However, STAR-MPI all-to-all routine offers better performance than the native routine on this cluster.

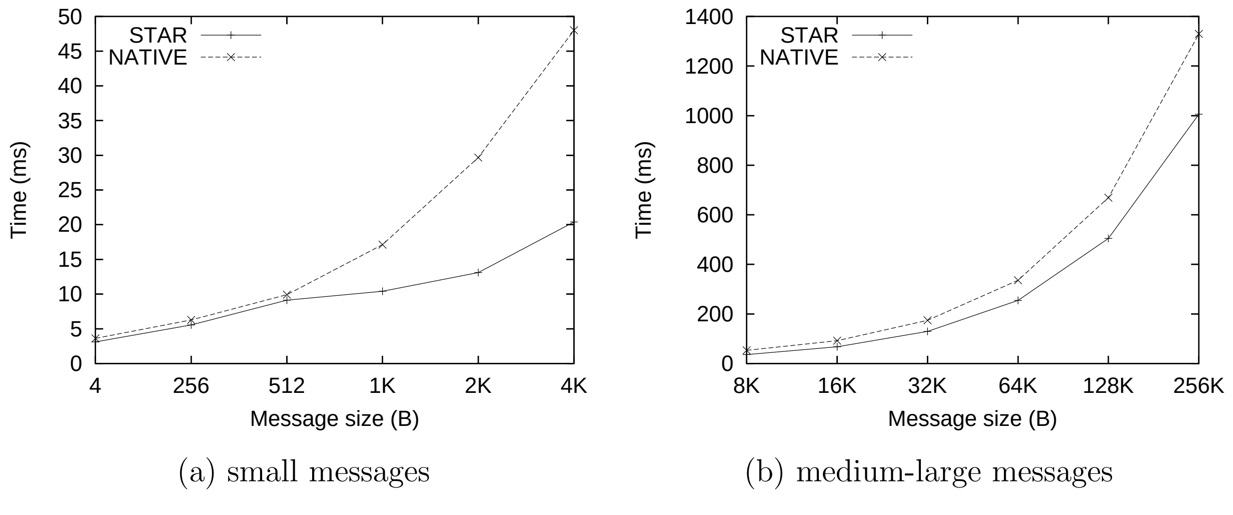


Figure 4.27: All-to-all micro-benchmark results on 128 processors (Lemieux), average per invocation time

The micro-benchmark results for *MPI_Alltoall* (500 iterations) on Lemieux are shown in Figure 4.27. The NATIVE legend denotes the performance of the native MPI routines. Part (a) of the figure shows the results for small messages while part (b) shows the results for medium/large messages. As shown in both parts of the figure, for all message sizes, STAR offers higher performance than NATIVE, especially as the message size increases. For example, when the message size is $256KB$, the respective completion times for STAR and NATIVE are 1005.6ms and 1329.5ms with STAR achieving a speedup of 32.3%. A similar performance trend is evident when experimenting on the FFTW application benchmark with $l = 5700$ and $nfft = 500$. The execution time for the benchmark using STAR-MPI is 184.3s as opposed to 212.8s for NATIVE, which is a 15.5% speedup.

Although the communication algorithms in STAR-MPI were designed for Ethernet-switched clusters, not for Lemieux, as shown in this experiment, STAR-MPI can improve the performance for other types of clusters. This demonstrates the robustness of the proposed DF technique.

4.4 Summary

The ability of MPI collective communication routines to adapt across platforms and applications is critical to construct efficient communication routines. Traditional library implementations of MPI, including LAM/MPI [44] and MPICH [52], fall short of such adaptability. The *delayed finalization of MPI collective communications* (DF) approach overcomes this limitation: it allows platform-specific algorithms to be included in the library and selects the best communication algorithm for a collective operation in a given platform and/or application configuration. STAGE-MPI and STAR-MPI integrate the two components of a DF technique: an algorithm repository of different communication algorithms (platform-specific and platform-unaware) that can potentially achieve high performance in different situations and an automatic algorithm selection mechanism, which is based on the AEOS [82] empirical approach, to select the best algorithm for a given situation. STAGE-MPI considers the architectural information of the platform to produce at the library installation time efficient MPI collective routines that significantly out-perform the ones used in LAM/MPI [44] and MPICH [52]. One limitation of STAGE-MPI routines is that they only adapt to the platform but not applications. The behavior of collective operations in MPI applications was studied, and it was found that an important aspect of application behavior, process arrival pattern, must be considered in the development of efficient MPI collective routines. The results of the study indicated that the performance evaluation of collective communication algorithms in the context of the application on the platform is necessary to achieve high performance. These results are considered in the development of the STAR-MPI library, which empirically measures the performance of communication algorithms in the context of both the application and platform. STAR-MPI routines are more efficient, achieve a higher level of software adaptability, and significantly out-perform the routines in traditional MPI libraries as well as the ones generated by STAGE-MPI in many cases.

The results in this chapter support the thesis of the dissertation: it is possible to develop an adaptive MPI library whose collective routines can adapt to platforms and applications, and the performance of MPI collective library routines can be significantly improved by incorporating platform/application specific communication algorithms in the library and making the library routines adaptive to the platform/application.

CHAPTER 5

CONCLUSION & FUTURE WORK

5.1 Conclusion

Realizing MPI collective communication routines that can achieve high performance across platforms and applications is challenging. In this research, the *delayed finalization of MPI collective communication routines* (DF) approach is proposed and investigated. By maintaining for each collective operation an extensive set of communication algorithms that may include both platform/application specific and platform/application unaware algorithms, and postponing the selection of the algorithms until the platform and/or application are known, DF based MPI libraries such as STAGE-MPI and STAR-MPI offer significantly better performance to applications than traditional MPI libraries such as LAM/MPI and MPICH. This demonstrates the effectiveness and practicality of the DF approach in realizing efficient MPI collective routines across platforms and applications.

5.2 Future Work

5.2.1 Performance Measurement Scheme for STAGE-MPI

One limitation of STAGE-MPI system is that it tunes MPI collective routines for Mpptest [32] programs, which may not select the best communication algorithm for real applications because the application context is unavailable. Note that the performance results of any timing mechanisms that are based on Mpptest-like schemes give a reasonable (but not highly accurate) estimate of how well an algorithm would perform in an application. A potential performance refinement opportunity is to use performance measurement schemes that somehow can reflect aspects of the application behavior. For example, to account for the impact of process arrival patterns at MPI collective call sites, in applications where a major collective routine is preceded by either a computation or another collective routine, users can measure the computation delay or the gap between the two collective routines. Such information can be then used to synthesize a performance measurement scheme (program) reflecting to some degree that particular program behavior. The new scheme can be used in STAGE-MPI to get more accurate performance estimates of the communication algorithms, which may result in selecting better communication algorithms.

Another related interesting issue is the impact of different timing mechanisms on the performance of the tuned collective communication routines. One potential study is to examine the impact of different timing mechanisms on one-to-all, one-to-many, all-to-one, and many-to-one types of collective operations as well as on different types of collective algorithms including *globally coordinated*, *store-and-forward*, and *directly communicating* (Section 4.2.5 details such algorithms).

5.2.2 Extending the DF Libraries

A potential future direction is to extend the DF libraries to other platforms with different nodal and networking architectures, such as systems with SMPs (or dual-core processors) or systems connected by Infiniband. Although STAGE-MPI and STAR-MPI can currently function properly on these platforms, they can only utilize the platform-unaware communication algorithms in the tuning process. To achieve higher performance, algorithms that are optimized for one or more parameters of these platforms may need to be developed. For example, in clusters of SMPs, memory bandwidth might be a performance factor, which may require developing algorithms that are specific to SMPs and can efficiently realize the memory bandwidth.

5.2.3 Process Arrival Pattern Sensitive Collective Algorithms

In this thesis, the process arrival patterns in MPI applications were studied. It was found that the arrival patterns can have a significant impact on the performance of collective communication algorithms and that the characteristics of arrival patterns must be considered in the development of efficient MPI collective routines. For example, as described in Section 4.2.5.1, many collective algorithms can be classified as *globally coordinated* algorithms, which include all phased algorithms such as the all–to–all ring, pair, and bruck algorithms [77, 7]. For these algorithms, an imbalanced process arrival pattern may cause the processes to be unable to coordinate as they are designed to: some processes may start a new phase earlier while others are still finishing their assigned tasks in the current phase. Potential future work in this context is to develop collective communication algorithms that are sensitive to process arrival patterns. These algorithms will attempt to cope with the negative effects of imbalanced arrival patterns.

5.2.4 Model Based DF Approach

The limitation of the empirical approach is that it takes a very long time to decide the best algorithm since all algorithms must be executed and measured. Using a performance model, the performance of different algorithms can be predicted analytically, which eliminates the execution and measurement overheads in the algorithm selection process: the efficient algorithm can be determined instantly. As discussed earlier, the challenge for developing a model based DF library is the development of analytical performance models that can predict the performance of different algorithms with sufficient accuracy.

REFERENCES

[1] D. H. Bailey, T. Harris, R. Van der Wigngaart, W. Saphir, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0,"Technical Report NAS-95-010, NASA Ames Research Center, 1995.

[2] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert, "Pipelined Broadcasts on Heterogeneous Platforms," *IEEE Transactions on Parallel and Distributed Systems*, 16(4):300–313, 2005.

[3] O. Beaumont, L. Marchal, and Y. Robert, "Broadcast Trees for Heterogeneous Platforms," *IEEE Transactions on Parallel and Distributed Systems*, 2005.

[4] G. D. Benson, C. Chu, Q. Huang, and S. G. Caglar, "A Comparison of MPICH Allgather Algorithms on Switched Networks," *In Proceedings of the 10th EuroPVM/MPI 2003 Conference*, Venice, Italy, pages 335–343, September 2003.

[5] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel, "Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology," In *Proceedings of the ACM SIGARC International Conference on SuperComputing*, pages 340–347, 1997.

[6] S. Bokhari, "Multiphase Complete Exchange: a Theoretical Analysis," *IEEE Transactions on Computers*, 45(2), pages 220–229, February 1996.

[7] J. Bruck, C. Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient Algorithms for All-to-all Communications in Multiport Message-Passing Systems," *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, November. 1997.

[8] F. Cappello and D. Etiemble, "MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks," SC00: High Performance Networking and Computing Conference, 2000.

[9] P.H. Carns, W.B. Ligon III, S.P. McMillan, and R.B. Ross, "An Evaluation of Message Passing Implementations on Beowulf Workstations," In Proceedings of the 1999 IEEE Aerospace Conference, March 1999.

[10] J. Cohen, P. Fraigniaud, and M. Mitjana, "Scheduling Calls for Multicasting in Tree Networks," In *10th ACM-SIAM Symposium on Discrete Algorithms* (SODA '99), pages 881–882, 1999.

[11] W. E. Cohen and B. A. Mahafzah, "Statistical Analysis of Message Passing Programs to Guide Computer Design," In *Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences*, volume 7, pages 544–553, Kohala Coast, Hawaii, 1998.

[12] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Eicken, "LogP: Towards a Realistic Model of Parallel Computation," Principles Practice of Parallel Programming, pages 1–12, 1993.

[13] H. G. Dietz, T. M. Chung, T. I. Mattox, and T. Muhammad, "Purdue's Adapter for Parallel Execution and Rapid Synchronization: The TTL_PAPERS Design," *Technical Report*, Purdue University School of Electrical Engineering, January 1995.

[14] V. V. Dimakopoulos and N.J. Dimopoulos, "Communications in Binary Fat Trees," *International Conference on Parallel and Distributed Computing Systems*, Florida, pages 383–388, September 1995.

[15] A. Faraj and X. Yuan, "Message Scheduling for All-to-all Personalized Communication on Ethernet Switched Clusters," *19th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Denver, Colorado, April 4-8, 2005.

[16] A. Faraj, X. Yuan, and P. Patarasuk, "A Message Scheduling Scheme for All-to-all Personalized Communication on Ethernet Switched Clusters," *IEEE Transactions on Parallel and Distributed Systems*, 2006, accepted.

[17] A. Faraj, P. Patarasuk, and X. Yuan, "Bandwidth Efficient All-to-all Broadcast on Switched Clusters," *The 2005 IEEE International Conference on Cluster Computing (Cluster 2005)*, Boston, MA, September 27-30, 2005.

[18] A. Faraj, P. Patarasuk, and X. Yuan, "Bandwidth Efficient All-to-All Broadcast on Switched Clusters," Submitted to *IEEE Transactions on Parallel and Distributed Systems.*

[19] A. Faraj and X. Yuan, "An Empirical Approach for Efficient All-to-All Personalized Communication on Ethernet Switched Clusters," *The 34th International Conference on Parallel Processing (ICPP)*, Oslo, Norway, June 14-17, 2005.

[20] A. Faraj and X. Yuan, "Automatic Generation and Tuning of MPI Collective Communication Routines," *The 19th ACM International Conference on Supercomputing (ICS)*, Cambridge, Massachusetts, June 20-22, 2005.

[21] A. Faraj and X. Yuan, "Characteristics of Process Arrival Patterns for MPI Collective Operations," Submitted to SC 2006.

[22] A. Faraj, X. Yuan, and D. Lowenthal, "STAR-MPI: Self Tuned Adaptive Routines for MPI Collective Operations," *The 20th ACM International Conference on Supercomputing (ICS06)*, Queensland, Australia, June 28-July 1, 2006.

[23] A. Faraj and X. Yuan, "Communication Characteristics in the NAS Parallel Benchmarks," In *Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, pages 724–729, November 2002.

[24] P. Patarasu, A. Faraj, and X. Yuan, "Pipelined Broadcast on Ethernet Switched Clusters," *The 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, April 25-29, 2006.

[25] P. Patarasu, A. Faraj, and X. Yuan, "Pipelined Broadcast on Ethernet Switched Clusters," Submitted to *IEEE Transactions on Parallel and Distributed Systems.*

[26] X. Yuan, S. Daniels, A. Faraj, and A. Karwande, "Group Management Schemes for Implementing MPI Collective Communication over IP-Multicast," *The 6th International Conference on Computer Science and Informatics*, pages 76-80, Durham, NC, March 8-14, 2002.

[27] FFTW, available at http://www.fftw.org.

[28] S. M. Figueira, "Improving Binomial Trees for Broadcasting in Local Networks of Workstations," *VECPAR'02*, Porto, Portugal, June 2002.

[29] M. Frigo and S. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 3, pages 1381–1384, Seattle, WA, May 1998.

[30] M. Golebiewski, R. Hempel, and J. L. Traff, "Algorithms for Collective Communication Operations on SMP Clusters," *In The 1999 Workshop on Cluster-Based Computing held in conjunction with 13th ACM-SIGARCH International Conference on Supercomputing (ICS'99)*, pages 11–15, Rhodes, Greece, June 1999.

[31] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," In *MPI Developers Conference*, 1995.

[32] W. Gropp and E. Lusk, "Reproducible Measurements of MPI Performance Characteristics," In *MPI Developers Conference*, 1995.

[33] J. Han and C. Han, "Efficient All-to-All Broadcast in Switch-Based Networks with Irregular Topology," *The Fourth International Conference on High-Performance Computing in the Asia-Pacific Region*, pages 112–116, Beijing, China, May 2000.

[34] S. Hinrichs, C. Kosak, D.R. O'Hallaron, T. Stricker, and R. Take, "An Architecture for Optimal All–to–All Personalized Communication," In *6th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 310–319, Cape May, New Jersey, June 1994.

[35] M. Jacunski, P. Sadayappan, and D.K. Panda, "All-to-All Broadcast on Switch-Based Clusters of Workstations," *Proceedings of 1999 International Parallel Processing Symposium*, pages 325–329, San Juan, Puerto Rico, April 1999.

[36] S. L. Johnsson and C. T. Ho, "Optimum Broadcasting and Personalized Communication in Hypercubes", *IEEE Transactions on Computers*, 38(9):1249-1268, September 1989.

[37] L. V. Kale, S. Kumar, and K. Varadarajan, "A Framework for Collective Personalized Communication," *International Parallel and Distributed Processing Symposium (IPDPS'03)*, page 69a, April 2003.

[38] A. Karwande, X. Yuan, and D. K. Lowenthal, "CC-MPI: A Compiled Communication Capable MPI Prototype for Ethernet Switched Clusters," In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'03)*, pages 95–106, San Diego, CA, June 2003.

[39] R. Kesavan, K. Bondalapati, and D.K. Panda, "Multicast on Irregular Switch-based Networks with Wormhole Routing," In *Proceedings of the Third International Symposium on High Performance Computer Architecture (HPCA'97)*, pages 48–57, Febuary, 1997.

[40] 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 Proceeding Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, pages 131–140, May 1999.

[41] H. Ko, S. Latifi, and P. Srimani, "Near-Optimal Broadcast in All-port Wormhole-routed Hypercubes using Error Correcting Codes," *IEEE Transactions on Parallel and Distributed Systems*, 11(3):247–260, 2000.

[42] S. Kumar and L. V. Kale, "Scaling All-to-All Multicast on Fat-tree Networks," *The 10th International Conference on Parallel and Distributed Systems (ICPADS 2004)*, pages 205–214, Newport Beach, CA, July 2004.

[43] D. Lahaut and C. Germain, "Static Communications in Parallel Scientific Propgrams," In *Proceedings of the 6th International PARLE Conference on Parallel Architectures and Languages*, pages 262–276, Athens, Greece, July 1994.

[44] LAM/MPI Parallel Computing, available at http://www.lam-mpi.org.

[45] LAMMPS: Molecular Dynamics Simulator, available at http://www.cs.sandia.gov/~sjplimp/lammps.html.

[46] C. C. Lam, C. H. Huang, and P. Sadayappan, "Optimal Algorithms for All–to–All Personalized Communication on Rings and two dimensional Tori," *Journal of Parallel and Distributed Computing*, 43(1):3–13, 1997.

[47] M. Lauria and A. Chien, "MPI-FM: High Performance MPI on Workstation Clusters," In *Journal of Parallel and Distributed Computing*, 40(1): 4–18, January 1997.

[48] W. Liu, C. Wang, and K. Prasanna, "Portable and Scalable Algorithms for Irregular All–to–all Communication," The *16th International Conference on Distributed Computing Systems (ICDCS'96)*, pages 428–435, Hong Kong, May 1996.

[49] P.K. McKinley, H. Xu, A. Esfahanian and L.M. Ni, "Unicast-Based Multicast Communication in Wormhole-Routed Networks," *IEEE Transactions on Parallel and Distributed Systems*, 5(12):1252–1264, December 1994.

[50] The MPI forum, "Version 1.1 of the MPI Document," available at *http://www.mpi-forum.org/docs/mpi-11.ps*.

[51] The MPI Forum, "The MPI-2: Extensions to the Message Passing Interface," available at http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[52] MPICH – A Portable Implementation of MPI, available at http://www.mcs.anl.gov/mpi/mpich.

[53] Myricom, available at http://www.myri.com.

[54] NASA Parallel Benchmarks, available at http://www.nas.nasa.gov/NAS/NPB.

[55] H. Ogawa and S. Matsuoka, "OMPI: Optimizing MPI Programs Using Partial Evaluation," In *Supercomputing'96*, Pittsburgh, PA, November 1996.

[56] E. Oh and I. A. Kanj, "Efficient All-to-All Broadcast Schemes in Distributed-Memory Parallel Computers," *The 16th International Symposium on High Performance Computing Systems and Applications (HPCS'02)*, pages 65–70, Moncton, New-Brunswick, Canada, June 2002.

[57] Parallel NBody Simulations, available at http://www.cs.cmu.edu/˜scandal/alg/nbody.html.

[58] ParaDyn: Parallel Molecular Dynamics With the Embedded Atom Method, available at http://www.cs.sandia.gov/ sjplimp/download.html.

[59] F. Petrini, D. J. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8192 Processors of ASCI Q," *IEEE/ACM SC'03 Conference*, pages 55–72, Phoenix, AZ, November 2003.

[60] Pittsburg Supercomputing Center, available at http://www.psc.edu/machines/tcs.

[61] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. Dongarra, "Performance Analysis of MPI Collective Operations," *IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, Denver, CO, April 2005.

[62] R. Ponnusamy, R. Thakur, A. Chourdary, and G. Fox, "Scheduling Regular and Irregular Communication Patterns on the CM-5," *Supercomputing*, pages 394–402, 1992.

[63] A. Proskurowski, "Minimum Broadcast Trees," *IEEE Transactions on Computers*, c-30, pages 363–366, 1981.

[64] R. Rabenseifner, "A new optimized MPI reduce and allreduce algorithms," available at http://www.hlrs.de/organization/par/services/models/mpi/myreduce.html.

[65] R. Rabenseinfner, "Automatic MPI counter profiling of all users: First results on CRAY T3E900-512," In *Proceedings of the Message Passing Interface Developer's and User's Conference*, pages 77–85, Atlanta, GA, March 1999.

[66] I. Rosenblum, J. Adler, and S. Brandon, "Multi-processor molecular dynamics using the Brenner potential: Parallelization of an implicit multi-body potential," *International Journal of Modern Physics*, 10(1): 189–203, February, 1999.

[67] SCI-MPICH: MPI for SCI-connected Clusters, available at http://www.lfbs.rwth-aachen.de/users/joachim/SCI-MPICH/pcast.html.

[68] D.S. Scott, "Efficient All–to–All Communication Patterns in Hypercube and Mesh-topologies," *the Sixth Distributed Memory Computing Conference*, pages 398-403, Portland, OR, April 1991.

[69] S. Sistare, R. vandeVaart, and E. Loh, "Optimization of MPI Collectives on Clusters of Large Scale SMPs," In *Proceedings of SC99: High Performance Networking and Computing*, Portland, OR, November 1999.

[70] N.S. Sundar, D. N. Jayasimha, D. K. Panda, and P. Sadayappan, "Hybrid Algorithms for Complete Exchange in 2d Meshes," *IEEE Transactions on Parallel and Distributed Systems* 12(12): pages 1–18, December 2001.

[71] T.B. Tabe, J.P. Hardwick, and Q.F. Stout, "Statistical analysis of communication time on the IBM SP2," *Computing Science and Statistics*, 27: 347-351, 1995.

[72] T. Tabe and Q. Stout, "The use of the MPI communication library in the NAS Parallel Benchmark," *Technical Report CSE-TR-386-99*, Department of Computer Science, University of Michigan, November 1999.

[73] A. Tam and C. Wang, "Efficient Scheduling of Complete Exchange on Clusters," *the ISCA 13th International Conference on Parallel and Distributed Computing Systems*, pages 111–116, Durham, NC, August 2000.

[74] A. Tanenbaum, "Computer Networks," 4th Edition, 2004.

[75] H. Tang, K. Shen, and T. Yang, "Program Transformation and Runtime Support for Threaded MPI Execution on Shared-Memory Machines," *ACM Transactions on Programming Languages and Systems*, 22(4): 673–700, July 2000.

[76] R. Thakur and A. Choudhary, "All-to-all Communication on Meshes with Wormhole Routing," *8th International Parallel Processing Symposium (IPPS)*, pages 561-565, Canc'un, Mexico, April 1994 .

[77] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimizing of Collective Communication Operations in MPICH," *ANL/MCS-P1140-0304*, Mathematics and Computer Science Division, Argonne National Laboratory, March 2004.

[78] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra, "Automatically Tuned Collective Communications," In *Proceedings of SC'00: High Performance Networking and Computing*, pages 46-48, Dallas, TX, 2000.

[79] E. A. Varvarigos and D. P. Bertsekas, "Communication Algorithms for Isotropic Tasks in Hypercubes and Wraparound Meshes," *Parallel Computing*, 18(11): 1233–1257, 1992.

[80] J. S. Vetter and A. Yoo, "An empirical performance evaluation of scalable scientific applications," In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Baltimore, Maryland, 2002.

[81] J. Watts and R. Van De Gejin, "A Pipelined Broadcast for Multidimentional Meshes," *Parallel Processing Letters*, 5(2): 281–292, 1995.

[82] R. C. Whaley and J. Dongarra, "Automatically tuned linear algebra software," In *SuperComputing'98: High Performance Networking and Computing*, Orlando, FL, November 1998.

[83] F. Wong, R. Martin, R. Arpaci-Dusseau, and D. Culler, "Architectural Requirements of Scalability of the NAS Parallel Benchmarks," SC99: High Performance Networking and Computing Conference, 1999.

[84] Y. Yang and J. Wang, "Efficient all-to-all broadcast in all-port mesh and torus networks," *Proceedings of 5th IEEE International Symposium on High-Performance Computer Architecture (HPCA-5)*, Orlando, FL, pages 290–299, January 1999.

[85] W. Yu, D. Buntinas, and D. K. Panda, "Scalable and High Performance NIC-Based Allgather over Myrinet/GM," TR-22, OSU-CISRC, April 2004.

[86] E. W. Zegura, K. Calvert and S. Bhattacharjee, "How to Model an Internetwork," *IEEE Infocom '96*, pages 594-602, April 1996.

BIOGRAPHICAL SKETCH

Ahmad Faraj

The author was born in Kuwait on March 26, 1976. In the spring of 2000, he completed his Bachelors degree in Computer Science (CS) at Florida State University (FSU). Under the advisement of Prof. Xin Yuan, he obtained his Masters degree in Fall of 2002, also from the Department of Computer Science at FSU. He is currently enrolled in the CS doctoral program at FSU and continues to work with Prof. Yuan on different research projects. His research interests include MPI implementation, communication optimizations and communication algorithms, performance analysis/optimization/tuning, empirical optimization techniques, parallel programming and computing, high performance computing, clustering, and compilers. The author is a member of IEEE and ACM.