2005

ATCOM: Automatically tuned collective communication system for SMP clusters.

Meng-Shiou Wu
Iowa State University

ATCOM: Automatically Tuned Collective Communication System for SMP Clusters

by

Meng-Shiou Wu

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Ricky A. Kendall, Co-major Professor
Zhao Zhang, Co-major Professor
Brett M. Bode
Mark S. Gordon
Diane T. Rover

Iowa State University

Ames, Iowa

2005

Graduate College
Iowa State University

This is to certify that the doctoral dissertation of

Meng-Shiou Wu

has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Committee Member

Signature was redacted for privacy.

Committee Member

Signature was redacted for privacy.

Committee Member

Signature was redacted for privacy.

Co-major Professor

Signature was redacted for privacy.

Co-major Professor

Signature was redacted for privacy.

For the Major Program

DEDICATION

For my wife Mei-Ling, my son Yasuki, and my parents.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Conventional implementations of collective communications are based on point-to-point communications, and their optimizations have been focused on efficiency of those communication algorithms. However, point-to-point communications are not the optimal choice for modern computing clusters of SMPs due to their two-level communication structure. In recent years, a few research efforts have investigated efficient collective communications for SMP clusters. This dissertation is focused on platform-independent algorithms and implementations in this area.

There are two main approaches to implementing efficient collective communications for clusters of SMPs: using shared memory operations for intra-node communications, and overlapping inter-node/intra-node communications. The former fully utilizes the hardware based shared memory of an SMP, and the latter takes advantage of the inherent hierarchy of the communications within a cluster of SMPs. Previous studies focused on clusters of SMP from certain vendors. However, the previously proposed methods are not portable to other systems. Because the performance optimization issue is very complicated and the developing process is very time consuming, it is highly desired to have self-tuning, platform-independent implementations. As proven in this dissertation, such an implementation can significantly outperform the other point-to-point based portable implementations and some platform-specific implementations.

The dissertation describes in detail the architecture of the platform-independent implementation. There are four system components: shared memory-based collective communications, overlapping mechanisms for inter-node and intra-node communications, a prediction-based tuning module and a micro-benchmark based tuning module. Each component is carefully

designed with the goal of automatic tuning in mind.

CHAPTER 1. INTRODUCTION

1.1 Overview

Message passing is the de-facto standard for many parallel applications. The primary message passing library (MPL) in wide use today is based on the message passing interface (MPI) standard [28]. Message passing applications spend a significant amount of time in communications, including point-to-point communications, collective communications and synchronizations. A profiling study [38] showed that parallel applications spend more than eighty percent of transfer time in collective communications. The result suggests that optimizing collective communications is crucial for real world parallel applications, especially for communication intensive applications.

In general, developers assume a flat communication architecture for data exchange, e.g. the network is fully connected with equal bandwidth among any pair of nodes. However, modern computing systems have complex, hierarchical communication structures for which no single communications algorithm works well unconditionally. This has forced application developers to implement multiple versions of their programs so that they may adapt to various systems. In some cases, application developers optimize their program for one system, but the programs are adapted to another system of a very different hardware architecture. To maintain the productivity of programmers and end users, it is desirable to have portable implementations of the MPI library that hide the complexity of the underlying hardware. Such an implementation must be capable of automatic tuning to couple with the large tuning space of modern hardware architecture.

One would expect that vendor implementations of MPI libraries should give the best performance. This is not always true, as shown in our experimental results. Surprisingly even vendors have trouble supporting all of their own infrastructures. Consider the varieties in the IBM SP system series. The architecture of some new generations of SP systems is very different from the previous ones. In this case, vendors may support highly efficient implementations for a few platforms, but in general the implementations are sub-optimal.

Portable MPI implementations, such as MPICH or LAM/MPI [74], are another choice for developers on those architectures. As those portable implementations must support a wide range of target systems, which have an almost infinite combination of interconnection strategies and network software stacks, they cannot be hand-tuned to the extent of the vendor versions. These libraries typically support the set of algorithms that give the best average performance.

In short, application developers or end users cannot be expected to manually tune either their applications or the underlying MPI systems to fully exploit the available computational power and network capabilities. These problems motivate the need for a system that can automatically select the best available implementations and their configuration for optimal performance on a given set of hardware and software resources.

The goal of automatic tuning collective libraries is to provide mechanisms where optimal communication algorithms can be selected by the system rather than hand tuned by the MPL users. Such a system must have a good set of implementations and tuning mechanisms to select the right implementation and produce the optimal configuration for a given computational resource. The existing approaches assume that each node has only one processor, and communications are inter-node, point-to-point communications; the collective communications and tuning mechanisms are all developed based on this "one processor per node" assumption. However, when a cluster is composed of SMP nodes, the complexity of designing automatically tuned collective libraries increases by more than one dimension. Point-to-point communication is no longer the best communication mechanism, since the SMP architecture provides shared memory for communications within an SMP node. Programming models to design collective communications on SMP architectures must be defined and interactions between communication layers must be identified in order to design tuning mechanisms. The characteristics and performance model of collective communications on SMP clusters must be developed to provide predictions and to determine appropriate algorithmic choices. Those problems must first be solved in order to build an automatic tuning systems for SMP clusters.

Given the fact that many systems in the top 500 list [75] are SMP clusters, we expect research that addresses these issues to provide a foundation to design a practical automatic tuning system for collective communications on SMP clusters and will be a significant contribution to the community.

1.2 Parallel Architectures and Clusters

Parallel computing technologies have allowed the peak speeds of high-end supercomputers to grow at a rate that has exceeded Moore's Law, and improving the computing power of parallel systems has been a major research topic for decades. Although parallel computers provide much high computation power than desktop computers, the cost of a propriety parallel computer is usually very high and is affordable only to major companies and government or academic research institutes. As an alternative to propriety parallel computers, building cluster computing systems is a cost-effective alternative for increased computational power.

In the last few years, cluster computing technologies have advanced to the extent that a cluster can be easily constructed using heterogeneous compute nodes, running an arbitrary operation system, and connected by different kinds of networks. This makes it possible for most research laboratories and universities to have their own cluster computing systems. Any university department or research laboratory can now build their own clusters that meet their computational demands.

There is no precise definition of cluster. "The term cluster, can be applied both broadly (any system built with a significant number of commodity components) or narrowly (only commodity components and open source software)" [89]. We use the broad definition of cluster in this dissertation. Different kinds of parallel computing systems are discussed in this section.

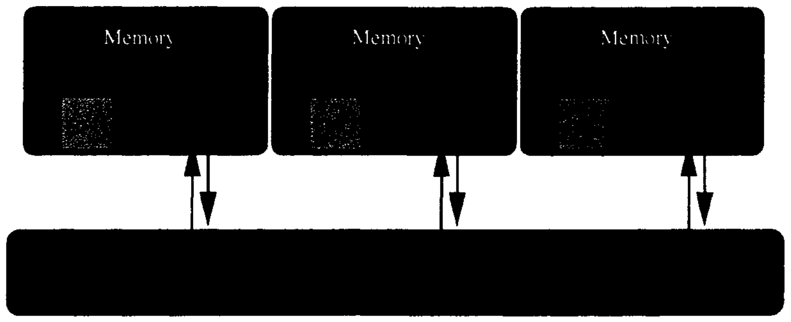


Figure 1.1 Block diagram of a distributed memory system.

1.2.1 Distributed Memory Systems

From a hardware perspective, the simplest approach to construct a parallel computing system is the distributed memory model. In this approach, separate compute nodes are connected by a network; each compute node has its own memory, CPU, network interface card (NIC), Operating System, etc. This type of system is the most common parallel computer systems since they are easy to assemble. Figure 1.1 depicts the basic building blocks of a distributed memory system. The propriety distributed systems such as IBM SP, Cray T3D or T3E may have special-purpose hardware and the cost of those systems are usually very high.

There are several projects that exploit the use of the low cost and high performance of commodity microprocessors to build distributed memory systems (sometimes refereed as NOWs, networks of workstations). The most famous are Beowulf clusters [90, 91], which are built from commodity parts and two of them reached top 100 supercomputer systems in 2000.

The communications between compute nodes is done through the interconnection network; one processor sends its message through the NIC (Network Interface Card) into the interconnection network, then another processor receives the message from the interconnection network via the NIC. Figure 1.2 shows the communications through the interconnection network between compute nodes, one node sends three messages to three nodes.

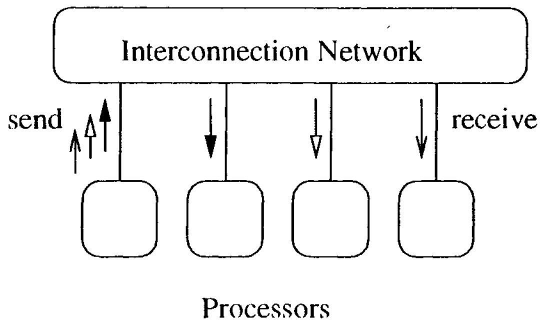


Figure 1.2 Communications through intercommunication network.

1.2.2 High Performance Interconnection Network

The interconnection network is a critical component of the computing technologies. Although the computational speed distinguishes high-performance computers from desktop systems, the efficient integration of compute nodes with interconnection networks also has a significant impact on the overall performance of parallel applications.

The interconnect networks commonly used in high performance computers include Gigabit Ethernet [53, 54], Myrinet [47], Quadrics [41, 42], and Infiniband [43]. Each provides a certain level of programmability, raw performance and the integration with the operation system. For example, InfiniBand provides multi-casting; Quadrics can utilize shared memory for collective communications. Since every high performance network has certain strengths of its own, utilizing a certain characteristic of a high performance network for communications is also an important research topic [40, 44, 45, 46, 48, 49, 50, 51, 52, ?].

1.2.3 Shared Memory Systems

In a shared memory system, the memory is placed into a single physical address space and supporting virtual address spaces across all of the memory. Figure 1.3 depicts the diagram of a shared memory system. Data in a shared memory system are available to all of the CPUs through load and store instructions. Because access to memory is not through network operations as in distributed memory systems, the latency to access memory is much lower. However, one major problem with shared memory system is cache coherence. Each CPU has

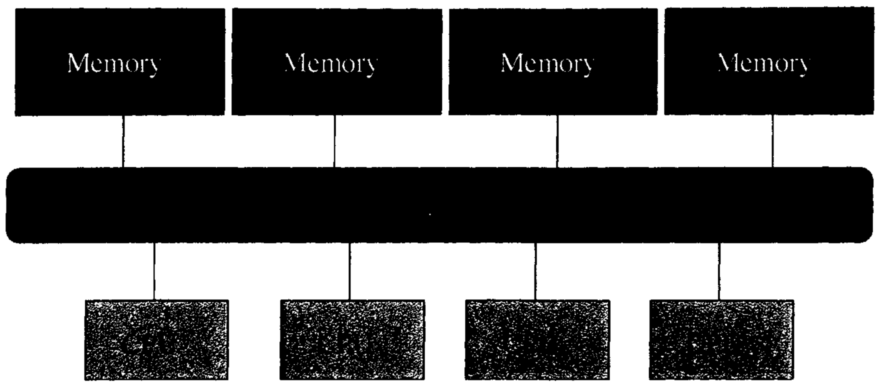


Figure 1.3 Block diagram of a shared memory system.

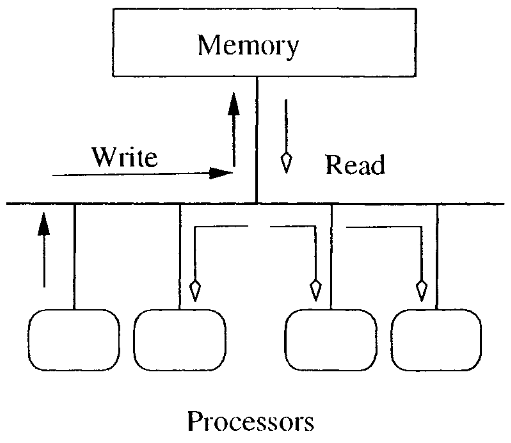


Figure 1.4 Communications through shared memory within an SMP node.

its own cache, and the mechanisms to keep data coherent in both cache and shared memory (as if cache were not present) may require additional hardware and can hinder application performance. There are two major types of shared memory machines; uniform memory access (UMA) and cache coherent nonuniform memory access (CC-NUMA) [89].

Shared memory systems usually have quite modest memory bandwidths. However, with the increase of processors in a shared memory system, some processor may be starved for bandwidth. "Applications that are memory-access bound can even slow as processors are added in such systems" [89]. For a low to medium end shared memory system, there are usually 2 to 16 processors in a system. A high-end system may have more than one hundred processors.

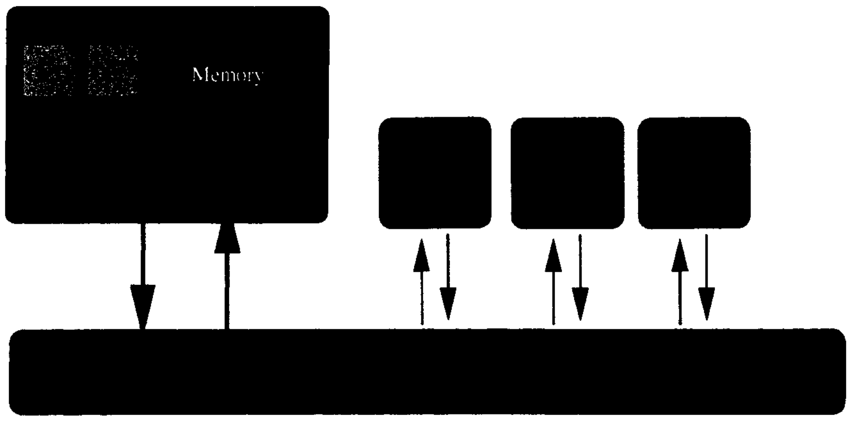


Figure 1.5 Block diagram of an SMP cluster.

For communications between the processors within the same compute node (intra-node communications), if they are processed through the interconnection network, there is no difference from the communications between SMP nodes. The communications can also be processed through shared memory, as shown in figure 1.4. Once a process writes data into shared memory, all the other processes can read that data concurrently. However, this approach for communications has its limitation; with the increase of number of processors, without careful coordination, the bus can become the performance bottleneck and the cost of cache coherence will increase.

1.2.4 The Architecture of SMP Clusters

Figure 1.5 depicts an SMP cluster in the most generic form. Compute nodes are connected through an interconnection network. Within an SMP node a shared system bus connects the memory with processors, serving as the medium for intra-node communications. The inter-node connection is a collection of links and switches that provide a network for all the nodes in the cluster. If an SMP node is an MPP (massively parallel processor) node, the communication between an SMP processor and the interconnection network are through a communication assist (CA). The communication assist acts as another CPU that is dedicated to handle communications. If an SMP node is a workstation, the communications from an SMP processor to the interconnection are a through network interface card (NIC).

The SMP architecture adds the hierarchical characteristic into the cluster environments, not only in the memory subsystem but also in the communication subsystem. For the memory system, there is remote memory (memory on another SMP node) and local memory (memory within an SMP node) as in any kind of clusters. For communications, there are inter-node communications - the communications between nodes, and intra-node communications - the communications within a node.

The communications between compute nodes in an SMP cluster are the same as inter-node, point-to-point based communications. The intra-node communications can have different approaches, as described earlier. Many MPI implementations use shared memory send and receive; in that case there are two layers of point-to-point communications: through the interconnection network and through the shared memory.

In this dissertation, our experiments were tested on both propriety clusters and commodity built PC clusters. The SMP clusters are of medium size (each SMP node has 2 to 16 processors), and the networks on the testing platforms include IBM's propriety network and Myrinet.

1.3 Parallel Programming Models

As discussed in the last section, there are different kinds of parallel systems. Different programming models can be developed for different types of parallel architectures. However, it is rarely the case that we develop a programming model for only one particular architecture; we usually develop a programming model based on a "virtual architecture" that can be applied on different parallel systems. For example, the programming model of MPI assumes the underlying network is fully connected, but on a real parallel system the interconnection network may have a certain topology. A theoretical model, parallel random access model (PRAM), assumes constant memory access time even in a real shared memory machine. The memory access time for each memory unit may be different. In this section, we describe the programming models for different parallel architectures.

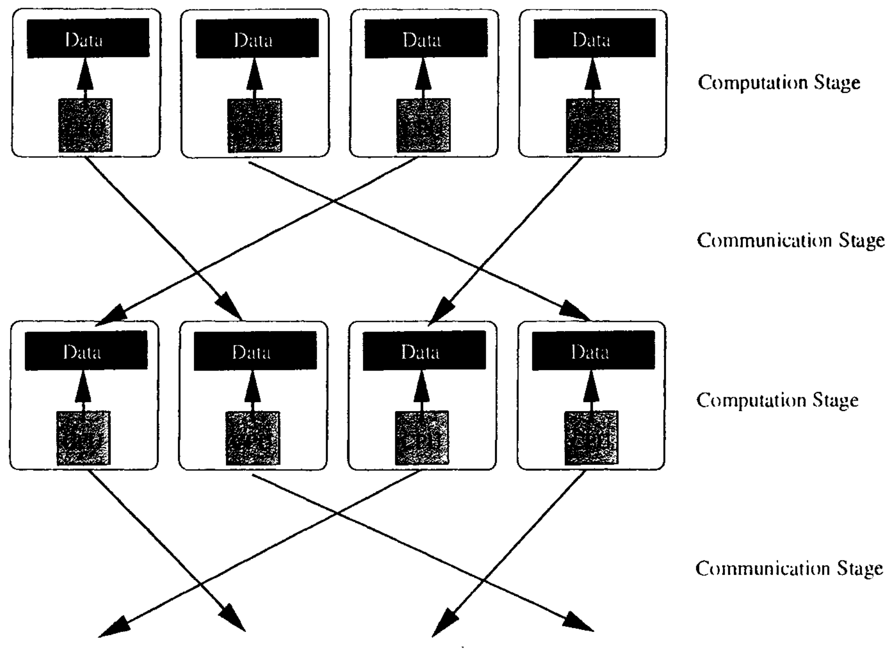


Figure 1.6 The programming model on the distributed memory systems.

1.3.1 Parallel Programming on Distributed Memory Systems

The most commonly used programming model on the distributed memory systems is to use message passing. In this programming model the execution of a parallel application is divided into computation stages and communication stages. Each compute node processes a certain amount of computation, followed by data exchange through message passing, then the computation-communication stages are repeated until the application terminates. This is shown in Figure 1.6. MPI (message passing interface) [28] and PVM (parallel virtual machine) [92] are the two most commonly used environments for this type of programming.

Although the MPI style of programming is currently the dominant style of designing distributed memory parallel applications, it is regarded as the assembly level of parallel programming since data (computation) must be explicit divided and moved (coordinated) between compute nodes. To simplify the programming efforts and reduce the burden of a programmer, several parallel programming toolkits and languages have been developed such as Global Arrays (GA) [93], High Performance Fortran (HPF) [69] and SHMEM [94].

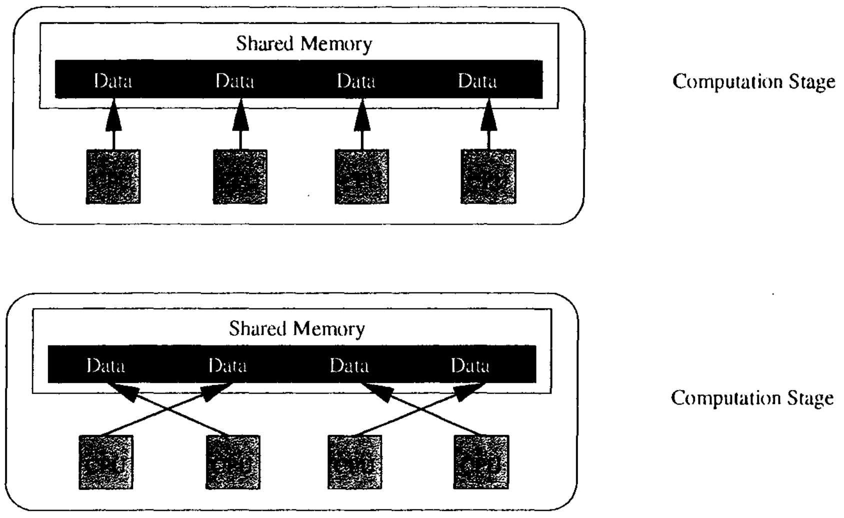


Figure 1.7 The programming model on the shared memory systems.

1.3.2 Parallel Programming on Shared Memory Systems

The PRAM model is the most commonly used model to design parallel programs on shared memory systems; the model assumes that memory access time is constant for every memory location in the shared memory. In the PRAM programming model the whole computation sequence is divided into smaller computation stages; during each computation stage each process works on a certain, unique data segment, with careful coordination of reading or writing data. This programming model is shown in Figure 1.7.

Pthreads [95] is one programming language (library) that can be used to implement algorithms developed with PRAM model. However, Pthreads is also regarded as assembly level shared memory parallel programming, since the data segments for each thread must be carefully defined and the computational sequence must be carefully coordinated. For scientific applications, a different programming language, OpenMP [70] was developed to leverage the burden of writing scientific applications on shared memory systems.

1.3.3 Parallel Programming on SMP Clusters

The hierarchical memory and communication structures on SMP clusters provides interesting opportunities for improving the performance of parallel applications, thus different pro-

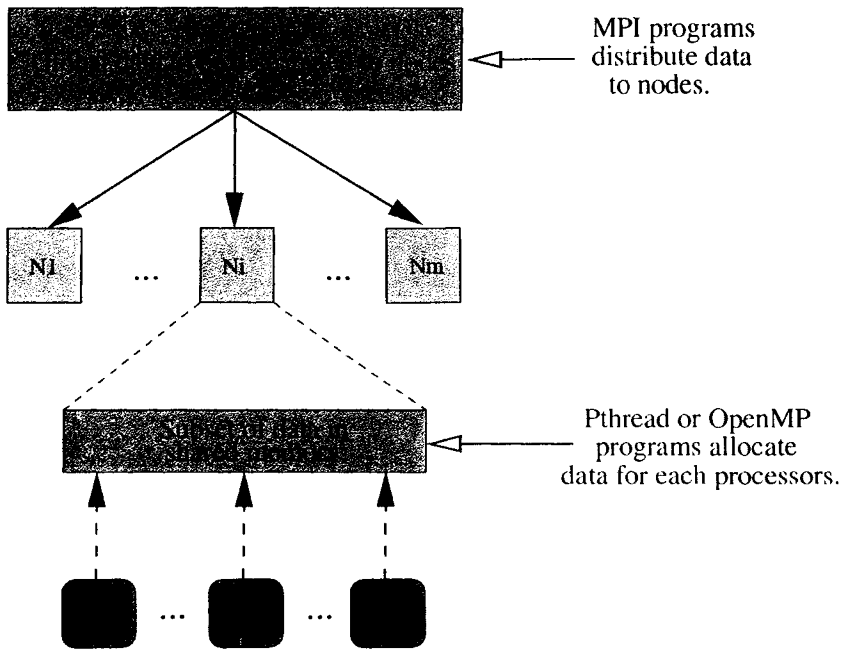

MPI programs distribute data to nodes.
N1
...
Ni
...
Nm
Pthread or OpenMP programs allocate data for each processors.

Figure 1.8 Mixed Mode Programming Model for SMP clusters.

gramming models have been developed to take advantage of the SMP architecture. The programming model for uni-processor clusters, such as the PRAM programming model, the BSP model [2], or the distributed memory programming model for MPI programming, all assume no communication hierarchy and thus cannot take full advantage of the SMP architecture. An emerging programming model for SMP clusters to date is to use MPI to design the inter-node layer of parallel applications, partition data and distribute data into the distributed memory on different compute nodes. After data is in place within an SMP node, OpenMP is used for an additional layer of parallelism.

A different approach is to use Pthreads for parallelization within an SMP node as one of our experiments on mixed mode programming [112]. The detail of this research is shown in the appendix. However, Pthreads programming is primarily targeted at systems programming, and there is only one full implementation for the Fortran interface. Application developers are generally encouraged to use OpenMP for the shared memory layer of parallelization. Figure 1.8 shows this mixed mode programming model.

Although we can use MPI plus OpenMP for programming on an SMP cluster, it is still an

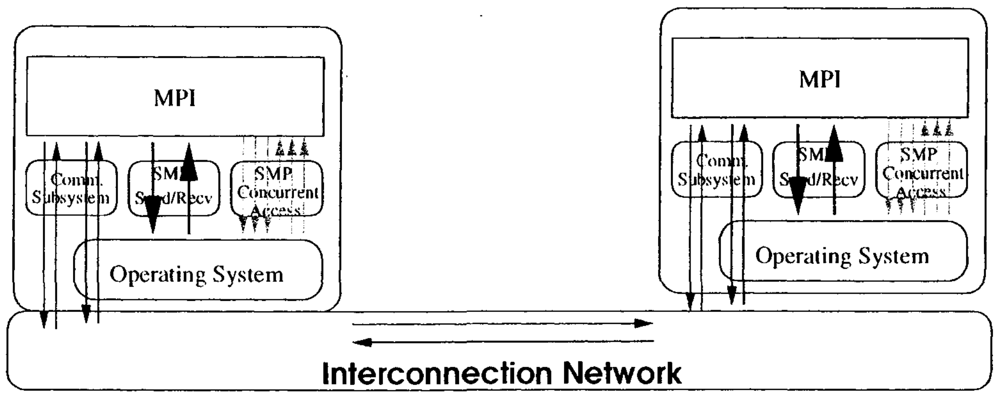


Figure 1.9 Block diagram of the MPI communication types on SMP cluster.

open problem as which programming style, pure MPI or mixed MPI and OpenMP, leads to better performance [84, 85, 86]. Moreover, mixing OpenMP and MPI programs requires redesign of many pure MPI applications, which in reality is error prone and has large development costs. Besides the mixed mode programming model, there are at least two programming model for SMP clusters: SIMPLE [87] designed by Bader *et al.* and Multi-Protocol Active Messages [83] designed by Lumetta *et al.*. Both require redesign and recoding of parallel applications.

The alternative for existing MPI applications is to take advantage of the SMP architecture via an SMP aware MPI implementation. If we can provide an MPI library that automatically utilizes the underlying SMP architecture for communications, existing MPI applications can gain improved performance without the need to make modifications.

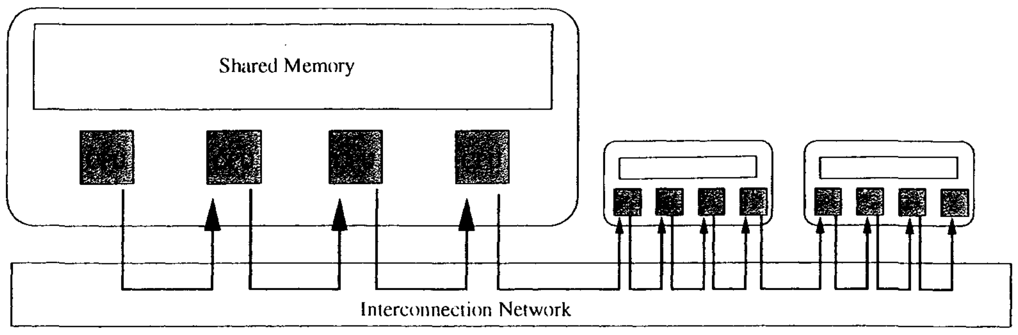
There are some existing MPI implementations that optimize send and receive using the SMP architecture for communications: MPICH2 [68], the MPI implementation by Protopopov and Skjellum [32], another MPI implementation for SMP clusters by Takahashi *et al.* [33] and IBM's MPI implementation. This optimization improves the performance of point-to-point communications, but is too simple for collective communications and does not take full advantage of the SMP architecture. We explain the deficiencies of this approach in detail in the next chapter.

1.4 MPI Collective Communications on SMP Clusters

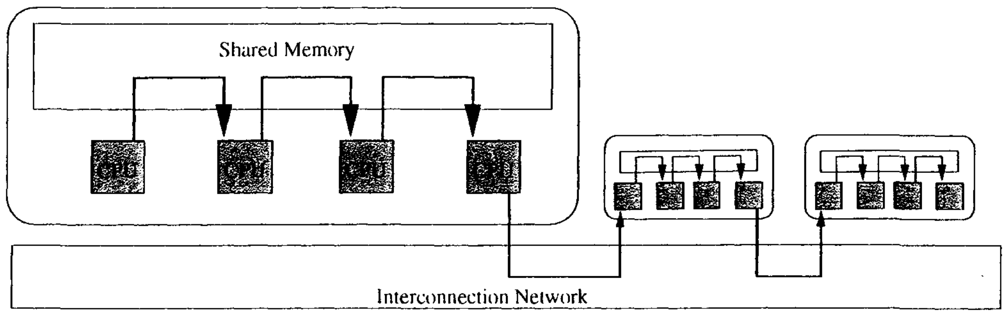
Since mixed mode programming (MPI plus OpenMP or Pthreads) requires redesign and recoding of an application and is error prone, our approach is to improve the performance of parallel applications on SMP clusters by enhancing the MPI implementations, i.e., make MPI implementations SMP aware. There are several approaches to implement MPI communications on SMP clusters. Figure 1.9 shows different communication types for MPI. If the underlying communication subsystem supports OS-bypass [71, 96] capability (a message goes directly to the interconnection network without one extra copy to operating system buffers), then it is possible for the MPI implementation to to send a message to another process without the intervention of operating system. Without OS-bypass, a message still has to go through the operating system and it will require at least one copy before the message is sent into the network (e.g., a copy from "user space" to "kernel space"). For the communication within an SMP node, the implementations of shared memory send/receive or concurrent memory access functions are usually done through standard system calls, which makes operating system intervention unavoidable.

MPI is designed with "point-to-point" communications in mind, thus most algorithms for collective communication are also designed base on "point-to-point" communications. On SMP clusters, point-to-point communications can be either through the interconnection network, or through shared memory. The point-to-point based sequential broadcast on an SMP cluster is shown in Figure 1.10(a), in which every communication is through network. In Figure 1.10(b) the communications within an SMP nodes are through shared memory, and the communications between SMP nodes are through the interconnection network. In Figure 1.10(c) the inter-node layer communications are the same as in (a) and (b), but the communications within an SMP nodes use concurrent memory access. We will discuss the details of this mode of operation in the next chapter.

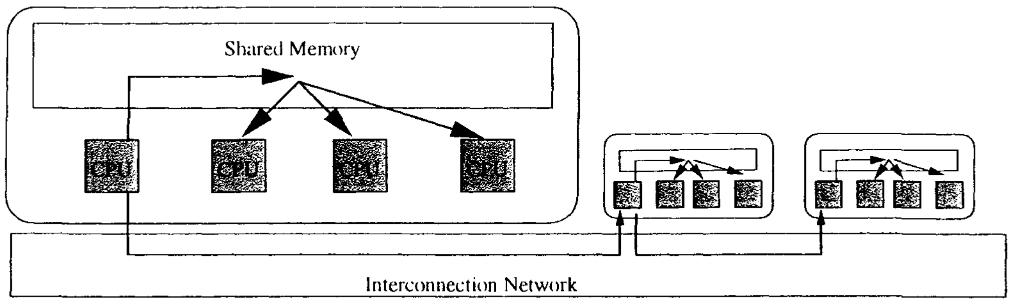
The strategy that uses concurrent memory access to design collective communications was first proposed by Sistare*et al.* [8] for several collective operations on an SUN SMP machine. However, there is no library that uses this approach to design all collective communications



(a) Point to point broadcast through the interconnection network.



(b) Point to point broadcast through the shared memory.



(c) Broadcast using concurrent memory access through the shared memory.

Figure 1.10 Three approaches to broadcast a message on SMP cluster.

that can be generally applied on different SMP clusters. How to utilize shared memory to design collective communications is one of the research topics in this dissertation.

1.5 Automatically Tuning Libraries

The complexity of current parallel architectures leads to different types of clusters. A cluster may consist of IBM SP2 nodes connected by Gigabit Ethernet, running the AIX Operating System, or may consist of Intel Xeon processors, connected by Myrinet, running Linux or FreeBSD. Such diversity of cluster architectures make it impossible to find a specific MPI implementation that is optimized for that particular cluster. There are more than ten collective operations in the MPI standard, and each operation can be implemented with many algorithms. As one can expect, an implementation may be the optimal choice for a certain platform under a certain setting, and at the same time it may not be the best choice for another platform under the same setting. This observation encouraged researchers to develop tunable collective communication library such as the CCL [24], which provides many implementations of a collective communication, and application developers must decide which implementation to use based on their target platforms and algorithms. However, expecting the administrator or application developers to manually tune the MPI for a cluster is impractical. The tuning mechanism in MPICH is to switch between different implementations based on message size, and this is too simple to provide optimal performance for different types of clusters. The natural engineering solution is to design mechanisms that can automatically tune the collective communications for optimal performance.

The methodology of automatic tuning was used to design ATLAS (Automatically Tuned Linear Algebra Software) [98] on uni-processor computers. In ATLAS different implementations of linear algebra functions are collected and tested, then the best implementation can be found base on the cache and memory architectures. When we apply the automatic tuning methodology on tuning collective communications, we can collect a set of good algorithms and implementations for different collective operations in the library. During the execution of an application when a collective operation is called, the best implementation of that operation is selected base on the runtime information such as message size, number of nodes, network topology.

Two existing automatic tuning MPI implementations are MagPIe [10, 11] and ACCT [14, 13]. ACCT, which is part of the FT-MPI [82] project, assumes each node has only one processor, and uses the strategy mentioned earlier. Although they proposed new implementations of collective communications and tuning strategies, many hand tuning processes are still required from an application developer.

1.6 Problem Description

The purpose of this research is to develop the foundation for an automatically tuned collective communication system on SMP clusters. The key problems in building such a system includes: developing approaches to take advantage of the SMP architecture for collective communications, exploring the performance model, and providing tuning strategies for the newly developed collective communications on SMP clusters.

The design methodology for the two existing systems is either too simple for SMP clusters (ACCT assumes only one layer of communication), or not an optimal choice for SMP clusters (MagPIe was targeting for clusters connected by a wide area network, WAN). Both are based on point-to-point communications. A fundamental question is, how to utilize the SMP architecture to design collective communications, and at the same time the optimizing techniques used must also be portable?

The novelty of this approach is the generic utilization of the SMP architecture as the foundation of the library. There are several building blocks that are crucial for the design and implementation of such as system.

1.6.1 Programming Model for Collective Communications within an SMP node

The first challenge in this research is to develop a generic programming model that allow us to design collective communications within an SMP node. The existing approaches in the literature optimize MPI_Send/MPI_Recv through shared memory and use implementations for inter-node collective communications directly. A few MPI implementations that utilize concurrent memory access implement only a small set of collective communications on a few specific platforms. For example, Sistare *et al.* [8] designed broadcast, reduce and barrier on a SUN cluster. Tipparaju *et al.* [9] design the same three collective communications for IBM SP clusters. With more complex parallel applications, more complex operations, such as scatter, gather, all-to-all, etc, must also be considered.

To our best knowledge, there are no generic guidelines on how to develop different collective operations on a SMP node. Moreover, when can the advantages of the SMP architecture (concurrent memory access) be utilized is also unclear. The first step of this research is thus to develop a generic programming model that allows us to design *all* collective communications with concurrent memory access features, and also explore the limitations of this approach.

1.6.2 Generic Overlapping Mechanisms for Inter-node/Intra-node Communications

An important issue that requires a generic approach in collective operations on SMP clusters is the mechanisms that allow overlapping between inter-node/intra-node communications. A platform specific approach was proposed by Tipparaju *et al.* [9]. The approach uses remote direct memory access (RDMA) for inter-node communications to overlap intra-node communications, which uses concurrent memory access. The functions for RDMA are provided by the IBM LAPI library [88] which is specific to the IBM platform and requires the IBM proprietary switch technology.

From the point of view of portability, such a platform specific approach is not favorable. There are several alternative approaches that may allow us to design overlapping mechanisms. We may use ARMCI [80], developed by Pacific Northwest National Laboratory, to replace RDMA in Tipparaju's approach, or use similar RDMA functionalities provided by the MPI-2 [67] standard. We can even use the most generic non-blocking communications for overlapping inter-node/intra-communications. The key issue in this problem is to determine an approach that can provide efficient and portable mechanisms to overlap inter-node/intra-node collective communications.

The generic programming model for designing collective communications within an SMP node and the generic overlapping mechanisms for inter-node/intra-node communications are the two key characteristics of SMP collective communications that distinguish them from the point-to-point based ones. These are normally platform specific approaches. If generic approaches can be developed, we can construct a portable high performance design for collective communications on SMP clusters.

1.6.3 Efficient Collective Communications Design based on the New Generic Programming Model

Current collective communication algorithms [19, 20, 21, 22, 23, 24, 15, 16, 17, 25, 26, 27] are designed for the inter-node layer, and are point-to-point based only. Some algorithms consider the features of SMP architecture [18]; the approach is also based on point-to-point communications, and no overlapping between inter-node/intra-node communications. While the above generic programming model can be developed, it is clear that we need new algorithms and implementations that take into consideration of overlapping mechanisms and the SMP architecture. The collective operations being investigated should not be limited to those three most often explored collective communications, but should also include other more complex operations such as scatter, gather, all-gather, all-to-all.

1.6.4 Performance Modeling

The existing performance models for collective communications, such as the Hockney model [1], LogP [3], LogGP [4] or parameterized LogP [10], assume a flat communication structure with point-to-point communications. Even a model such as parameterized LogP that takes the hierarchal communication structure into account, the overlapping of communications between two communication layers is still based on overlapping point-to-point communications.

With the new programming model, the collective communications are using both shared memory collective communications and overlapping mechanisms (mixed mode collective communications). What are the characteristics that distinguish them from point-to-point based collective communications on SMP clusters? What is the performance model that can describe this kind of collective communications? A new performance model is needed for the new programming model so we can evaluate an implementation without implementing it.

When the new performance model is developed, it can provide better understanding of the interaction between inter-node/intra-node communications, performance predictions of mixed mode collective communications, and the tuning strategies to reduce the amount of experiments required to tune for optimal performance.

1.6.5 A Micro-benchmarking Set for Collective Communications

There are many algorithms for a collective operation, and for an algorithm there are many possible implementations; each implementation may have several parameters to tune to achieve optimal performance. It is not practical to implement every possible implementation for a collective communication. Also, the amount of experiments to extract the optimal values for parameters during run time can be very large and this can hurt the performance of an application.

Instead of implementing every possible implementation then conducting all performance tuning during runtime, an off-line performance tuning tool can be used to select the implementations that may provide good performance and filter out unnecessary experiments. Based on the results of the off-line tuning, the runtime tuning system conducts experiments that can only be done when the runtime information is available. This performance tuning tool is a micro-benchmarking set for collective communications which should provide the information such as shared memory buffer size, number of pipeline buffers, performance prediction of an algorithms or implementation, testing range of a certain parameter.

Constructing a detailed and complete analysis of how to design micro-benchmarks for every collective communication can take from months to years; this dissertation will only provide a guideline for designing this micro-benchmarking tool.

1.6.6 Foundation for an Automatic Tuning System

The key element of an automatic tuning collective communication system is to efficiently select the best approach during run time. Two existing systems, MagPIe and ACCT, use different approaches for this purpose. For MagPIe, it uses parameterize LogP [10] model for performance prediction. The values of the basic parameters (latency l, overhead o, gate value g) for the model are extracted from experiments. During runtime MagPIe uses those values to predict the performance of a certain approach and selects the one with the best prediction. The approaches that use run time calculation can be applied in a WAN environment in which the latency between local network and wide area network are at least two orders of magnitude larger and the computation time as well as the cost of local communications can be subsumed into the cost of wide area communications. ACCT exhaustively tests different combinations of {*operation, message size, segment size, number of nodes, algorithm*} and extracts the combination with the best overall performance. Since there are infinite possible combinations of the five parameters, a reasonable testing range of each parameter is assigned by heuristic approaches. How the extracted parameter information is used during run time was not mentioned.

On the SMP clusters, the choice to implement a collective communication is not limited to point-to-point based approach. In this dissertation we will discuss the possible approaches to design collective communications on SMP clusters. The vast number of possible implementations also implies that the tuning mechanisms in ACCT or MagPIe can not be directly applied. We will also discuss how to utilize each developed mechanism into the structure of an automatic tuning sequence that can efficiently extract the optimal implementations.

1.7 Summary

Figure 1.11 outlines the existing generic approaches, existing platform specific approaches, and the proposed approaches that are in the dissertation. In summary, the existing generic approaches are inter-node collective communication algorithms, implementations and performance model for point-to-point based communications as well as automatic tuning strategies for uni-node clusters. Existing platform specific approaches are a limited set of collective operations on a few SMP clusters and a RDMA inter-node approach to overlap inter-node/intra-node communications. The proposed approaches in this dissertation include: generic approaches of shared memory collective communications, an overlapping mechanism on SMP clusters, a new performance model and several tuning strategies for mixed mode collective communications. Together these mechanisms provide the foundation to build a practical automatic tuning collective communication system for SMP clusters.

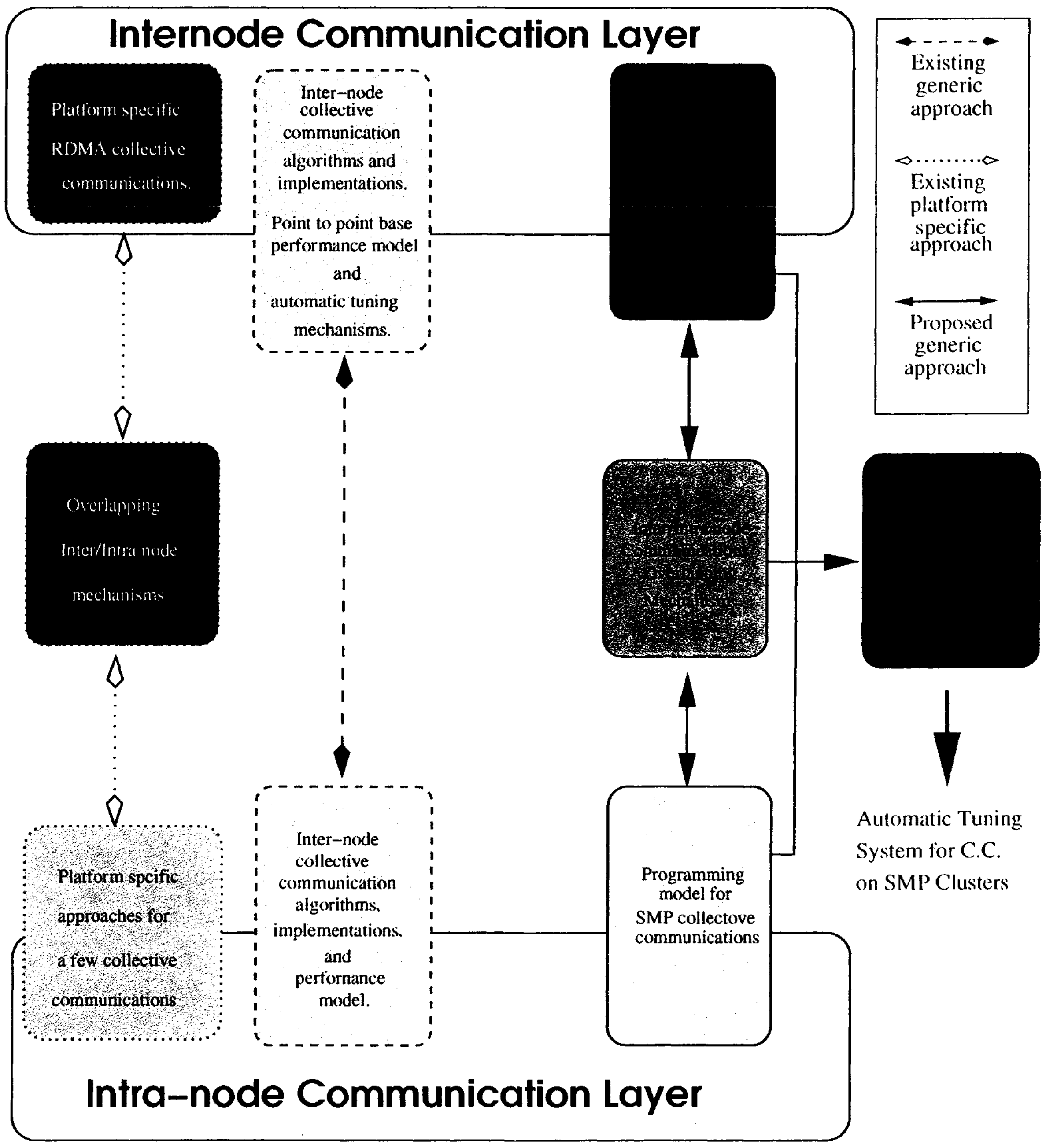

Internode Communication Layer
Platform specific RDMA collective communications.
Inter-node collective communication algorithms and implementations.
Point to point base performance model and automatic tuning mechanisms.
Existing generic approach
Existing platform specific approach
Proposed generic approach
Overlapping Inter/Intra node mechanisms
Platform spcific approaches for a few collective communications
Inter-node collective communication algorithms, implementations, and performance model.
Programming model for SMP collectove communications
Automatic Tuning System for C.C. on SMP Clusters
Intra-node Communication Layer

Figure 1.11 The existing generic approaches, the existing platform specific approaches and the proposed new generic approaches in diagram.

CHAPTER 2. TUNABLE COLLECTIVE COMMUNICATIONS FOR THE SMP ARCHITECTURE

2.1 Introduction

When designing collective communications for better performance within an SMP node, most MPI implementations focus on improving the efficiency of point-to-point based collective communications by implementing send and receive through shared memory. The feature of the SMP architecture, concurrent memory access, implies another possibility to improve the performance of collective communications. There are three approaches to implement collective communications within an SMP node. We introduced these three approaches in the previous chapter, and we discuss them in detail in this section.

2.1.1 Collective Communications Through the Interconnection Network

The first approach uses the intercommunication network to pass messages between MPI processes within a node. As long as each MPI process has access to NIC and collective communications are implemented using point-to-point communications, the collective communication implementations on the inter-node layer can be directly applied on the intra-node layer. The drawback of this approach is that, processes may have to compete with each other to gain access of NIC for communications. The experiment by Vadhiyar [13] shows that, the latency of performing a broadcast through NIC within an SMP node of 8 processors is much worse than that of broadcasting between 8 uni-processor nodes.

2.1.2 Collective Communications Through Shared Memory Send and Receive

The second and the third approaches use shared memory for collective communications. The second approach, the most commonly used approach by many MPI implementations, is to design send and receive through shared memory within an SMP node. As the first approach, the collective communication implementations on the inter-node layer can be used on the intra-node layer without any modification. The assumption of this approach is that, within an SMP node, the latency of sending a message from one processor to another processor through shared memory should be less than through the intercommunication network. By reducing the cost of sending or receiving a message, we should be able to reduce the overall cost of a collective communication. Several MPI implementations are of this type: MPICH2 [68], MPI implementation by Protopopov and Skjellum [32], another MPI for SMP clusters by Takahashi *et al.* [33] and IBM's MPI implementation. The optimizations of this approach usually focus on the problems such as how to design better algorithms base on shared memory send and receive, how to handle "flood" of messages, memory allocation [31, 36], etc.

However, optimizing just send and receive ignores the possible performance gain of using concurrent memory access on the SMP architecture, and sometimes it leads to bad performance due to extra memory copies caused by applying inter-node algorithms on the intra-node layer. Figure 2.1 shows the result of inter-node scatter of flat tree algorithm and binomial tree algorithm on an IBM cluster at National Energy Research Scientific Computing Center (NERSC); binomial tree algorithm performs better than chain tree algorithm. Figure 2.2 is the result of the same two algorithms implemented using shared memory send/receive on the intra-node layer. Flat tree performs better on the intra-node layer due to fewer extra memory copies. Analyzing the performance using the Hockney model [1] does not reveal the answer to this result since the cost of binomial tree is $logB * (\alpha + \beta * m)$, while sequential tree is $(B-1) * (\alpha + \beta * m)$. We will discuss how to use Hockney model to measure performance in a later section.

The reason for the poor performance of the binomial tree algorithm on the intra-node layer is due to memory copy overhead. The binomial tree algorithm used the "recursive doubling"

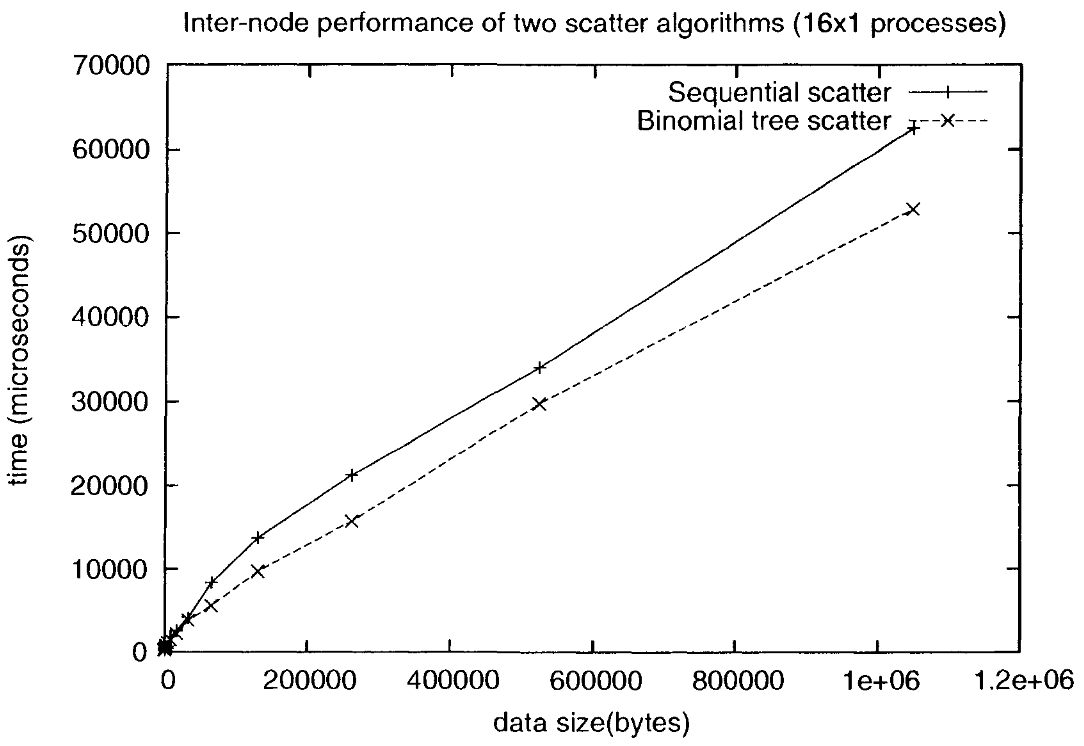


Figure 2.1 Performance of two scatter algorithms on the inter-node layer on the IBM SP cluster.

technique, which sends half of the scatter data to another process recursively so communication can be processed in parallel. Using this strategy, a message can be copied at most *logB* times before it reaches its destination, while a sequential algorithm a message needs at most one copy to reach its destination and incurs no extra memory copy.

We tested broadcast, scatter, gather, all-gather and all-to-all, since those collective operations can be implemented with a flat tree (sequential) algorithm and a binomial tree algorithm. Except for the broadcast operation, which does not require the recursive doubling technique, all other operations show similar results. When the data size is small, the recursive doubling technique provides better performance. As the data size increases, the performance of binomial tree algorithm degrades gradually, and the sequential algorithm eventually performs better. As for the broadcast, since the binomial tree broadcast does not incur any extra data movement overhead, it performs better then sequential algorithm for both inter-node and intra-node communications. This result is published in reference [114].

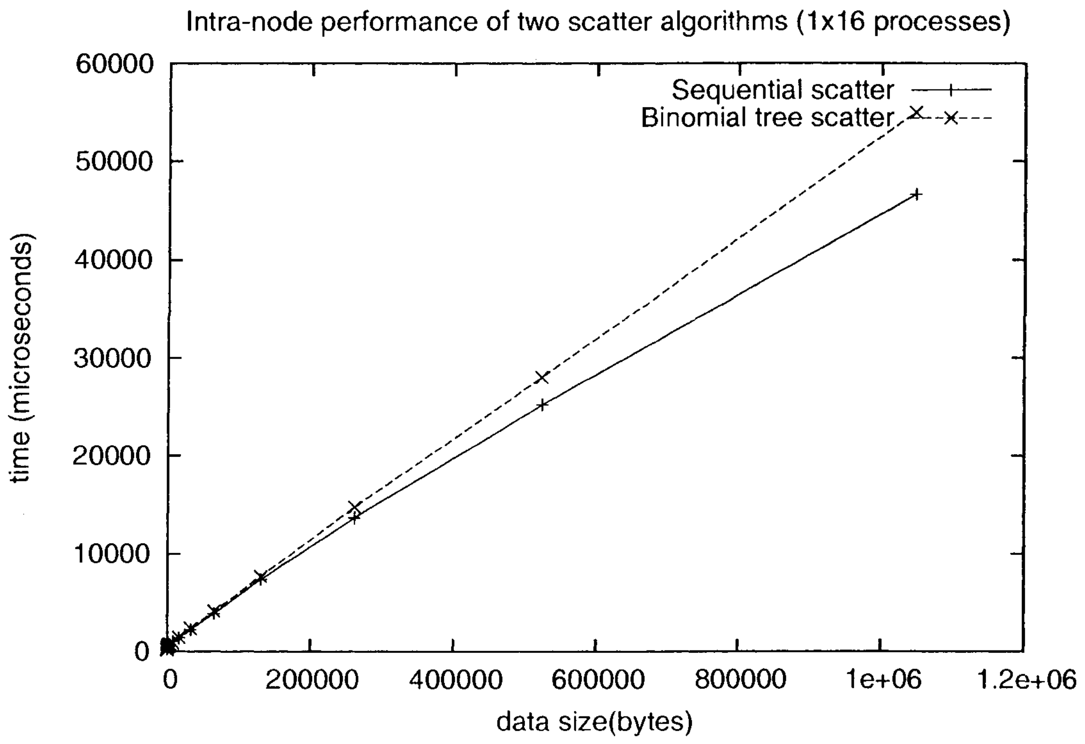


Figure 2.2 Performance of two scatter algorithms on the intra-node layer on the IBM SP cluster.

When shared memory send and receive are available, the latency on the intra-node layer is usually smaller than the latency on the inter-node layer, suggesting a hierarchy structure in communications. Some collective communication algorithms are developed base on such hierarchical structure. Golebiewski *et al.* [18] developed several collective communication algorithms base on the difference of latency on two communication layers within an SMP cluster. A different method is to directly apply MPI implementations for GRID computing on SMP clusters. In this method we can map the hierarchy structure of GRID communications (WAN / LAN) onto SMP communications (inter-node /intra-node). MPICH-G2 [29, 30] and MagPIe [10] are possible choices for this approach.

2.1.3 Collective Communications Using Concurrent Memory Access

The third approach to implement collective communications within an SMP node is to use concurrent memory access to design collective communications. This approach was first proposed by Sistare *et al.* [8] for three collective operations, broadcast, barrier and reduce, on an SUN SMP machine. A similar approach was proposed by Tipparaju *et al.* [9] on an IBM cluster. Again, they implemented the same three collective communications as in Sistare's approach.

The two existing methods implemented the set of collective operations that are frequently used, on a specific platform, and did not mention if it is possible to port their implementations to a different platform. This lead us to consider the following questions: (1). How can we design all collective operations within an SMP node using concurrent memory access features? (2). What is the limitation of designing collective communications with concurrent memory access? (3). How to design portable collective communications that use shared memory? (4). When porting these implementations to different platforms, what are the parameters we need to tune to achieve better performance?

2.2 The Generic Communication Model for SMP Clusters

To design portable shared memory collective communications, one approach is to investigate different SMP architectures, design the library for each available architecture, collect the programs into a library, and then compile the corresponding programs according to the target platform. Using this approach, the library is designed according to the characteristics of a certain platform and may provide very good performance. However, it is usually time consuming and also requires huge efforts to cover all kinds of platform.

Another approach is to explore the common characteristics and functionalities available across different SMP architectures, define parameters accordingly, and design communication library that the performance can be achieved by tuning those parameters on different platforms. In this approach, the time it takes to design the communication library can be reduced since the same implementation can be used across different platforms. However, the tuning time may be very long and the performance may not be as good as platform specific optimizations. In this dissertation we explore the potential of the second approach. We use a well understood communication model for SMP cluster as the base for our tunable collective communication

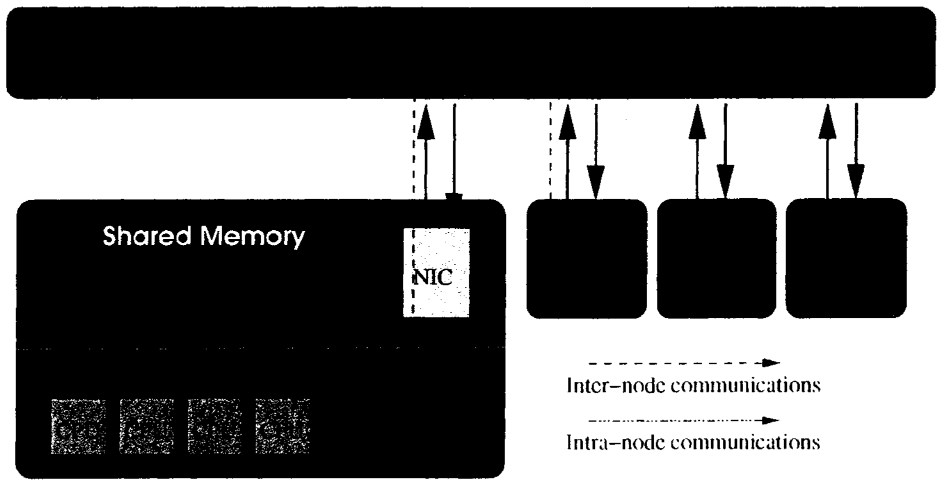


Figure 2.3 Generic Communication Model for SMP Clusters

library.

The communication model consists of two levels: the inter-node network communication and the intra-node shared-memory operations. On the intra-node layer, shared memory operations are implemented as follows. A shared-memory segment of *limited* size is allocated for communications between any two or more processors on the same node. Any communication within an SMP node is begun by the source process copying data from its local memory into the shared-memory segment, then the receiving processes copy data from the shared-memory segment to their local memories. The shared-memory operations are implemented with System V shared-memory functionality that are available on any UNIX-like systems, thus that are portable.

Within each SMP node, one processor (group coordinator) is in charge of scheduling communications with the other nodes. For communications between nodes we use standard MPI send/receive or non-blocking send/receive operations as a generic base. In this chapter we assume there is no overlapping between the two levels of communications, and all inter-node collective communications are implemented with blocking send/receive. In the next chapter we will discuss generic mechanisms to overlap inter-node/intra-node communications. Figure 2.3 outlines this generic communication model.

2.2.1 The Testing Platforms

Machine Type	*CPUs per node*	*Network type*	*MPI implementation*	*Testing MPI tasks*
IBM Power3	*16*	*IBM Propriety network*	*IBM MPI*	*16x16*
Intel Xeon	*2*	*Myrinet*	*MPICH 1.2.5.2*	*16x2*
Macintosh G4	*2*	*Myrinet*	*MPICH2 0.97*	*16x2*

Table 2.1 Three testing platforms.

Three testing clusters are listed in Table 2.1. The IBM SP system at the National Energy Research Scientific Computing Center is a distributed-memory parallel supercomputer with 380 compute nodes. Each node has 16 POWER3+ processors and at least 16 GBytes of memory, thus at least 1 GByte of memory per processor. Each node has 64 KBytes of L1 data cache and 8192 KBytes of L2 cache. The nodes are interconnected via an IBM proprietary switching network and run IBM's implementation of MPI. The Intel Cluster is located at Iowa State University. It consists of 44 nodes with dual Intel Xeon processors (88 processors). The nodes, running MPICH, are connected with Myrinet. The Macintosh G4 Cluster is located at the Scalable Computing Laboratory at Ames Laboratory. It is a 32-node "Beowulf" style cluster computer consisting of 16 single processor G4s with 512 MB RAM and 16 dual processor G4s with 1 GB RAM, all running Debian Linux. The G4 cluster uses Myrinet for the primary network communication layer. For our testing on the G4 we used only dual processor nodes, running MPICH-GM.

In this chapter we use only IBM SP cluster for our experiments since it is the only testing platform with more than two processors per node, and the approaches in this chapter can have very limited performance improvement on dual-processors clusters. In the later chapters we will show the experimental results on the other platforms as more complex methods are developed, and the performance improvement can be observed even on duel-processors clusters.

2.2.2 Notations

All the figures in this dissertation use the notation AxB to denote that the experiment uses A nodes, each with B MPI tasks. For example, 4x8 means we are using 4 nodes, each with 8 MPI tasks for the specific experiment. In this chapter the curves labeled with SHM mean we used our shared-memory implementation, and those label with XCC mean we utilized a combination of MPI for inter-node communications and shared-memory operations for intra-node communications. In the other chapters we use ATCOM to represent all our implementations.

2.3 Collective Communications on the Inter-node Layer

Many collective communication algorithms can be found in the literature such as CCL by Bala *et al.* [24], InterCom by Barnett *et al.* [21, 22], and the work of Mitra *et al.* on a fast collective communication library [20]. On the inter-node layer, we implemented the binomial tree algorithm and flat tree algorithm for four implemented collective communications: broadcast, scatter, gather and all-to-all. We also use scatter-allgather broadcast algorithm as an example of a two stages algorithm. We import it from MPICH 1.2.5 [15] with a small modification to make it work on the IBM SP cluster but keeping the basic algorithm intact. The detail of each tree algorithm and the designing issues of collective communications on this layer will be discussed in the next chapter.

All implementations in this chapter are implemented with blocking send/receive, and the tuning criteria on this layer are straightforward: find the algorithm with the best performance for a particular collective communications. Theoretical analysis does not always give the answer as discussed earlier. The best algorithm for an operation on a certain platform usually has to be found through experimentation.

2.4 Collective Communications on the Shared Memory Layer

To design tunable collective communications within an SMP node, we started by observing the operations of several major collective communications within a MPI communicator:

Broadcast: One sends and many receive the same data.

Scatter: One sends and many receive different data.

Gather: Everyone sends different data and one receives all data.

Allgather: Everyone sends different data and everyone receives all data.

Barrier: synchronization.

All-to-all: everyone sends different data and everyone receives different data.

From the collective operation perspective, when executing a collective communications, a process is in one of the following status:

(1). Sending a message to a another process.

(2). Receiving a message from another process.

(3). Sending a message to a group of process.

(4). Receiving messages from a group of process.

If these operations are implemented with point-to-point communications, (3) and (4) would be implemented in several stages using send and receive. On the other hand, (3) and (4) are what we can take advantage of SMP architecture.

According to our communication model, processes within an SMP node are communicating through a shared memory buffer of *limited* size. When the total communicating message size is smaller than this *limited* shared buffer size, all collective communications can be completed within one stage of concurrent memory access. If the communicating message size is larger than the shared memory buffer, we have to make proper use of pipelining similar as we use on the inter-node layer collective communications (detail of the inter-node pipelining strategies will be discussed in the next chapter).

Base on the above observation, we define the following basic shared-memory operations on a given SMP node:

(1) ***Sender_put()*** *: Sender puts message into a shared buffer.*

(2) ***Receiver_get()*** *: Receiver gets message from a shared buffer.*

(3) ***Pair_sync()*** *: Synchronization between two processes.*

(4) ***Group_get(p,m)*** *: A group of p processes gets a message of size m from a shared buffer.*

(5) ***Group_seg_get(p,m)*** *: A group of p processes gets a message from a shared buffer, each*

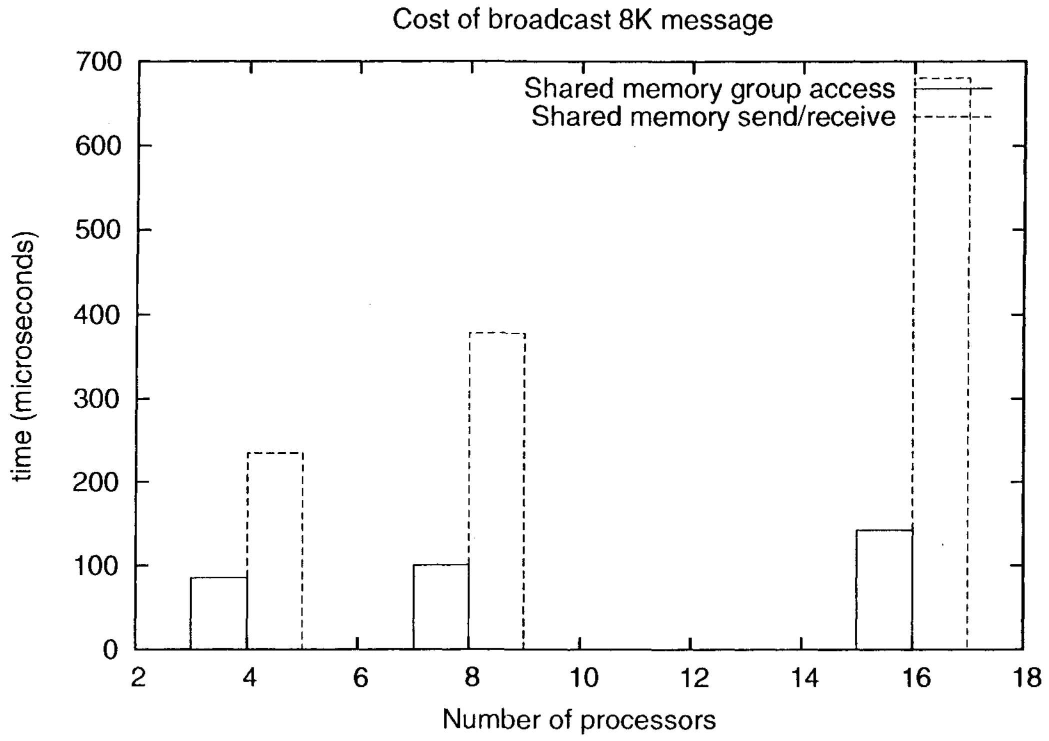


Figure 2.4 Cost of accessing data within an SMP node on the IBM SP cluster.

process gets its part of the data of size m.

(6) ***Group_seg_put(p,m)*** *: A group of p processes puts a message into a shared buffer, each process puts its part of the data of size m.*

(7) ***Group_sync(p)*** *: Synchronization among a group of p processes.*

(*Sender_put(), Pair_sync(), Receiver_get()*) is the minimum set of operations required for implementing collective communications on a shared-memory layer within an SMP node. A send and receive operation between processes in message passing can be replaced by (*Sender_put(), Pair_sync(), Receiver_get()*) within an SMP node. To utilize concurrent memory access, we have added four more operations that utilize this feature, and decompose collective communications into these seven basic shared-memory operations.

Our decomposition is based on the following observations: Figure 2.4 shows the performance of two broadcast approaches to access a data block of 8K. One approach uses the generic

C.C.	***(a) $m < B$***
1 Broadcast	*Sender_put(), Group_sync(), Group_get()*
2 Scatter	*Sender_put(), Group_sync(), Group_seg_get()*
3 Gather	*Group_seg_put(), Group_sync(), Receiver_get()*
4 All-to-all	*Group_seg_put(), Group_sync(), Group_seg_get() with pipeline*
5 Allgather	*Group_seg_put(), Group_sync(), Group_get()*

Table 2.2 Five collective communications decomposed into basic shared-memory operations when data size m is smaller than shared buffer size B

C.C.	***(b) $m > B$***
1 Broadcast	*Sender_put(), Group_sync(), Group_get() with pipeline*
2 Scatter	*Sender_put(), Pair_sync(), Receiver_get() with pipeline*
3 Gather	*Sender_put(), Pair_sync(), Receiver_get() with pipeline*
4 All-to-all	*pairwise-[Sender_put(), Pair_sync(), Receiver_get()] with pipcline*
5 Allgather	*Sender_put(), Group_sync(), Group_get() with pipeline*

Table 2.3 Five collective communications decomposed into basic shared-memory operations when data size m is smaller than shared buffer size B

shared-memory operations delineated in this chapter. This is one stage of (*Sender_put(), Group_sync(), Group_get()*). The second approach uses the vendor-based implementation and it is *logp* stages of shared-memory send and receive, p is the number of processors used within an SMP node. In all three cases the latency of shared memory operations are smaller than *logp* stages of send and receive. Moreover, when the number of processes increase, the rate of run time increase is also much less than the second approach. However, this advantage can be used only up to a certain data size. When the data size increases, the hidden cost of (*Sender_put(), Group_sync()* , *Group_get()*) such as page faults, TLB misses, and cache coherence maintenance also increases and this approach loses its advantage. The results also suggest that the tuning strategy is to find the best buffer size for shared-memory operations and use them whenever optimal.

In Table 2.2 and 2.3, we outline five collective communications as examples to show how to decompose these collective communications into basic shared-memory operations.

This is certainly not the only way to decompose collective communications. If we assume there are "hot spots" as mentioned in Sistare's work [8], we may want to develop different algorithms to avoid concurrent access of certain portions of memory. For example, this may be important in the hierarchical NUMA memory on an SGI origin system. Our approach is to find the maximum size that can take advantage of concurrent memory access, so we do not take "hot spots" into consideration.

2.4.1 Analytical Tuning Criteria

There are three mechanisms for communications within an SMP node: our shared memory approach, using the vendor send/receive shared-memory operations, and using communication network. We give our theoretical analysis here as the guideline to indicate when to use one approach and when to switch to another one.

We use Hockney model [1] as the base to evaluate the performance of collective communications. Assume α is the startup latency, β is the inverse transmission rate, m is the message size, A is the number of SMP nodes in a cluster, B is the number of processors per SMP node, and p is the total number of processors.

The latency of sending a message of size m between two processes on the inter-node layer is $(\alpha + \beta * m)$. If an inter-node broadcast is done by binomial tree algorithm, then the latency is:

$$logA * (\alpha + \beta * m). \tag{2.1}$$

If it is done using the flat tree algorithm, the inter-node communication latency is:

$$(A - 1) * (\alpha + \beta * m). \tag{2.2}$$

Within an SMP node, the communication cost through the network, assuming there is no contention of NIC between processes, is:

$$logB * (\alpha + \beta * m) \tag{2.3}$$

using binomial tree algorithm, and

$$(B - 1) * (\alpha + \beta * m) \tag{2.4}$$

using flat tree algorithm.

If intra-node communication is done by using the shared-memory send and receive, then the communication cost is *logB*(binomial tree algorithm), or *B-1*(flat tree algorithm) stages of shared-memory send and receive.

If concurrent memory access to shared memory is possible, then the intra-node communication latency is k stages of:

$$(group\_op(single\_op) + sync\_op + group\_op(single\_op)). \tag{2.5}$$

The value of k depends on m and how a collective communication is implemented on the shared-memory layer. Since in this chapter we assume there is no overlapping between inter-node and intra-node communication, the total cost of a collective communication is just the sum of communication costs on the two layers.

Assuming we use binomial algorithm for broadcast operation, then the choice of a particular approach basically depends on the relative latency of the following:

(1) $logB * (\alpha + \beta * m)$,

(2) *logB* stages of shared-memory send and receive, or

(3) k stages of*group_op(single_op) + sync_op + group_op(single_op)*.

Clearly, when the data size is small enough that $k = 1$, or when we can use group operations to access the data such as in the broadcast algorithm, (3) is the best choice. If there are only two processors per SMP node ($B = 2$), or we can only use send and receive on the shared-memory layer such as in the scatter or gather algorithm with the large data size, then optimized shared-memory send/receive (2) will certainly help. When communication through the network is faster than through shared memory, (2) can be replaced by (1). If network communication is so fast that when the data size is small, even the time cost of $logB * (\alpha + \beta * m)$ is smaller than one stage of *group_op(single_op) + sync_op + group_op(single_op)*, all that is needed is (1).

It is clear from the above analysis that taking advantages of SMP architecture can have performance gains when the data size is small to medium (when $k = 1$). Vetter conducted experiments on several large-scale scientific applications [78, 79] and observed, "the payload

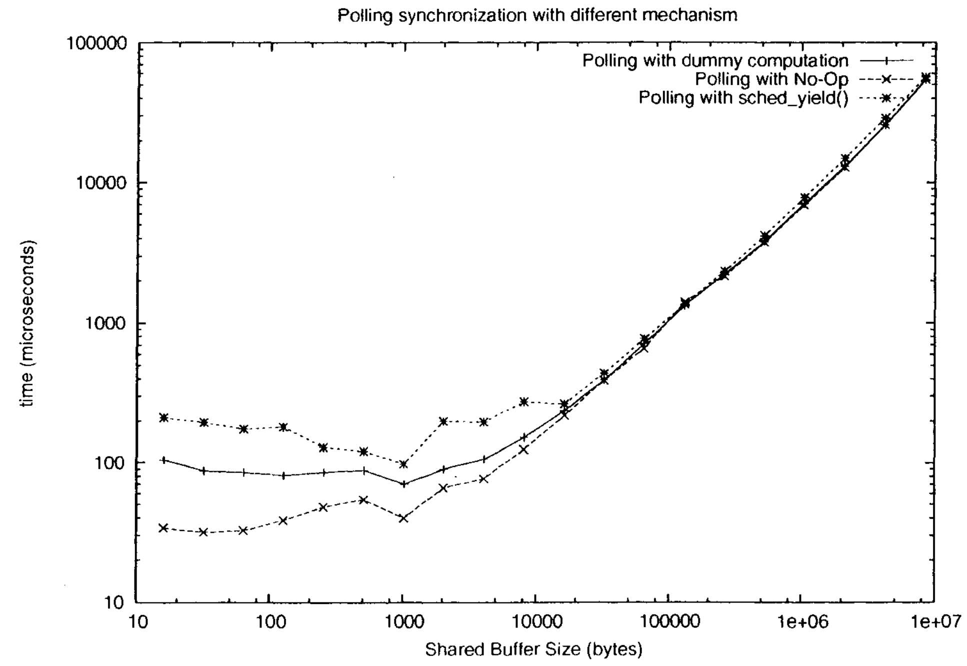


Figure 2.5 Performance of different synchronization schemes.

size of these collective operations is very small and this size remains practically invariant with respect to the problem size or the number of tasks." Based on the Vetter's observation, we believe that this approach should be considered as a way to improve the performance of MPI collective communications on SMP based clusters.

The theoretical formulations to predict the relative performance of (1), (2) and (3) are based on several basic parameters. There are tools such as the one provided by MagPIe [11, 12] to evaluate these parameters on different platforms. However, there is no tool to evaluate the contention of NIC by processes with the same SMP node, and it usually has to be measured by experiments. In this chapter we have chosen to compare them by experimentation so that we can shed more light on an analytical functional form to determine when to switch from one algorithm to another.

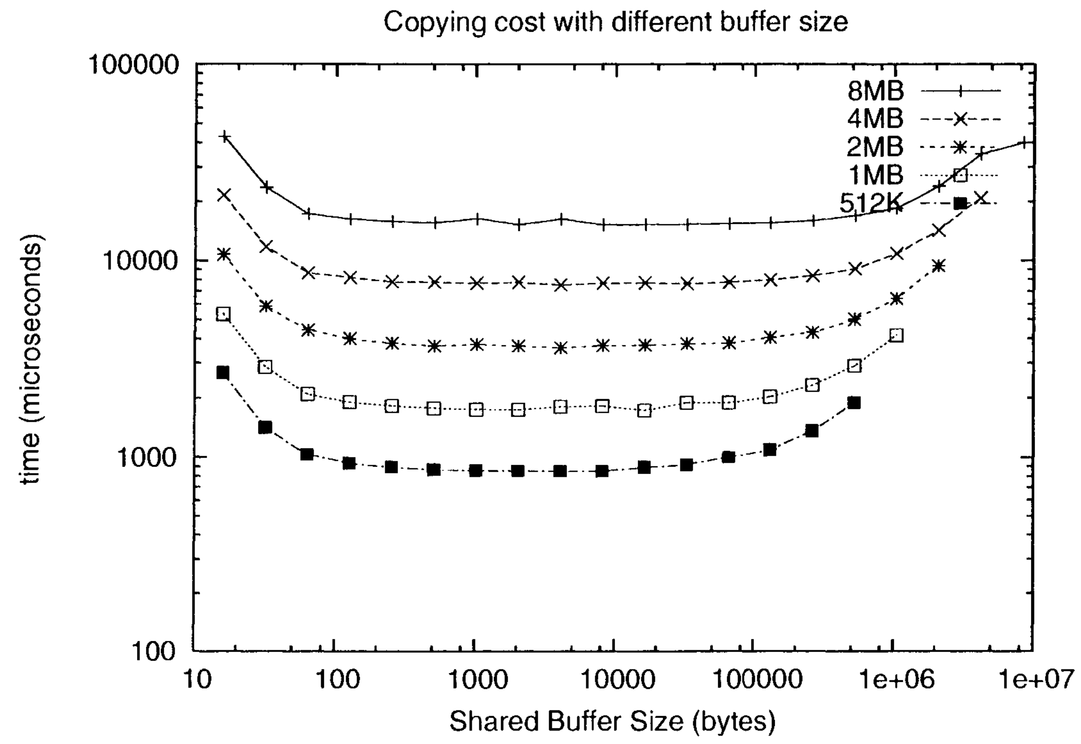


Figure 2.6 Shared buffer size and copying performance.

2.5 Tuning Parameters on the Shared Memory Layer

Base on the analysis, our turning strategy is to find the optimal buffer size such that a collective communication can be completed in one stage. When message size is larger than this buffer size, find the proper number of pipeline buffers to obtain better performance. When pair wise communications is the best choice for a collective communication, we switch to vendor's send/receive implementations if they are provided.

2.5.1 Synchronization Schemes

Several synchronization schemes are developed: polling with do nothing loop (no-op), polling with dummy computation, polling with *sched_yield()* function, and signal. We have tested these synchronization methods and found the following results. Using signal gives the worst performance; polling with *sched_yield()* provides good results only when shared memory buffer is large enough. Polling with no-op generally gives a good performance, and polling

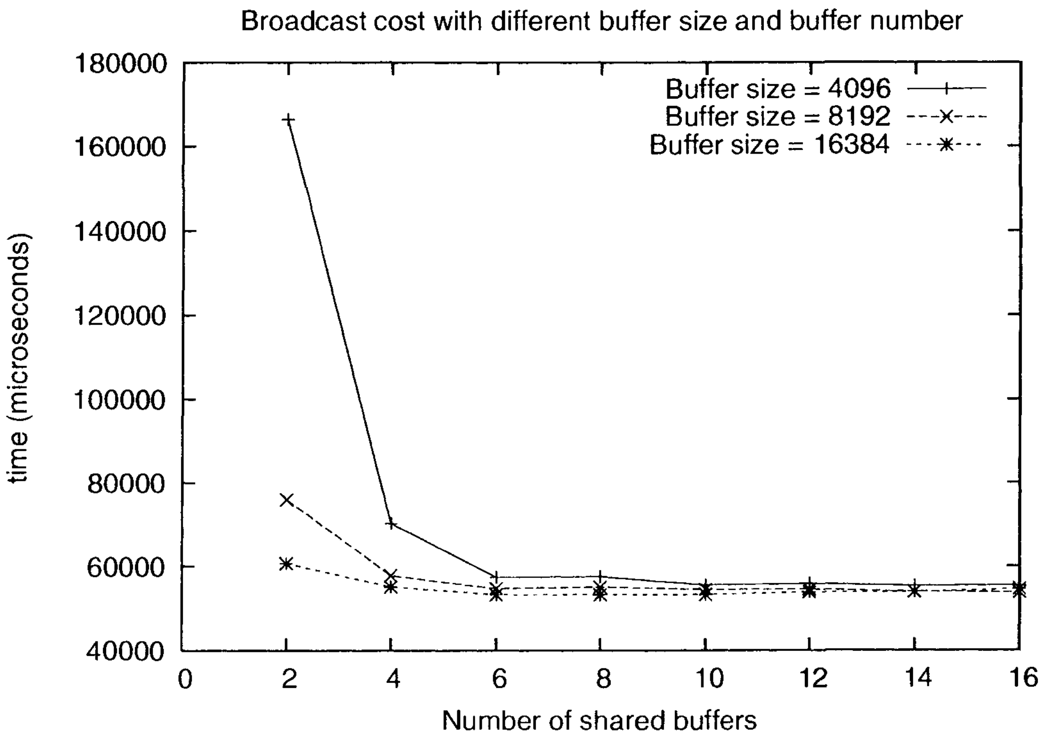


Figure 2.7 Performance of broadcast as a function of buffers on the IBM cluster.

with dummy computations gives slight better performance than polling with no-op when the shared buffer is large. Figure 2.5 shows the performance comparison of different synchronization schemes except for the signal approach. The experimental results in this dissertation all use polling with no-op.

2.5.2 Shared Buffer Size

With a huge buffer size the collective communication may be able to be completed in one stage, but hidden costs such as TLB misses may degrade performance. If we use a small buffer size, the sender and receiver may spend too much time in synchronization. To find a proper buffer size, we measured the cost of *sender_put()*. We tested data sizes from 1K, 2K, 4K up to 8MB, and for each data size we repeatedly copied data segments to buffer sizes of 16, 32, 64 bytes up to the test data size. Figure 2.6 shows the cost of copying data sizes of 512K to 8MB to shared memory through different shared-buffer sizes using log-log scale. The curve

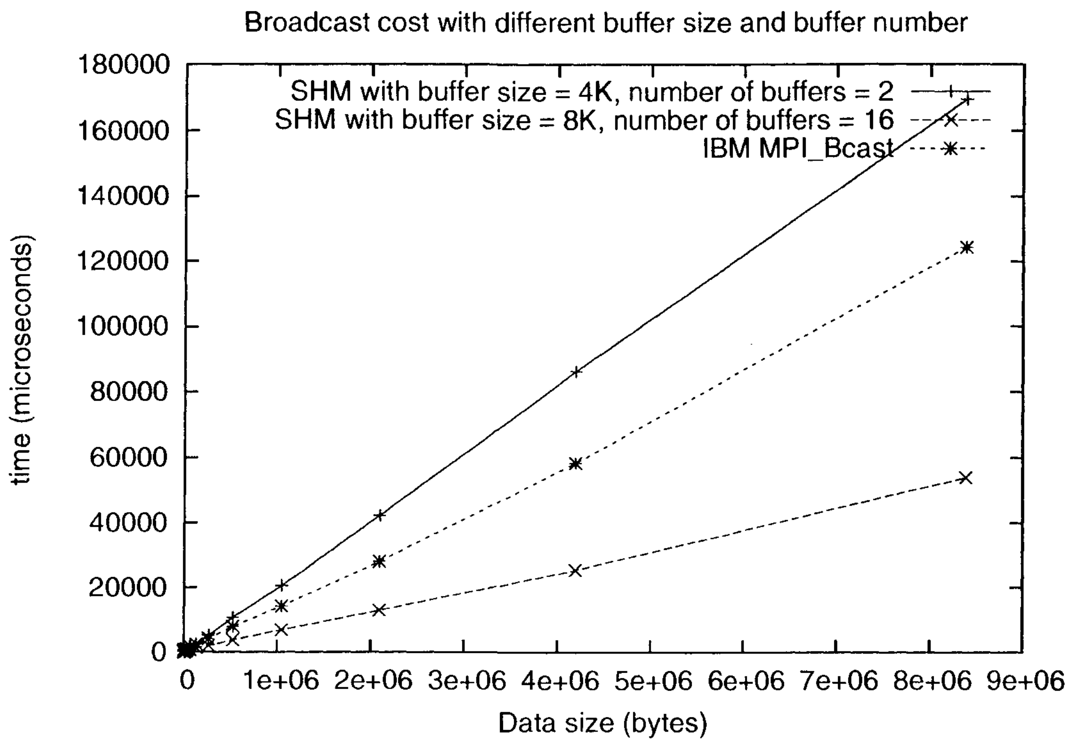


Figure 2.8 Performance of broadcast as a function of buffers

indicates a buffer from 4K to 16K is best for copying 8MB on our test system. We chose 8K as the shared-buffer size which generally gives a good performance.

2.5.3 Pipeline Buffers

When multiple buffers are used in a pipelined fashion, the number of buffers used affects performance. While two buffers are frequently suggested in the literature, it does not guarantee optimal performance. Figure 2.7 shows how we tuned buffer numbers for broadcast operations. Two buffers of size 8K do not perform as well as 2 buffers of size 16K. When we increase to 16 buffers, an 8K buffer size outperforms the other combinations for broadcast. Figure 2.8 is the performance of the same implementation, with and without tuning, against the vendor supplied MPI_Bcast() function. Without tuning (2 buffers of size 4K), the run time is 30% slower than the vendor implementation. After tuning, the run time is 50% faster than the vendor MPI_Bcast.

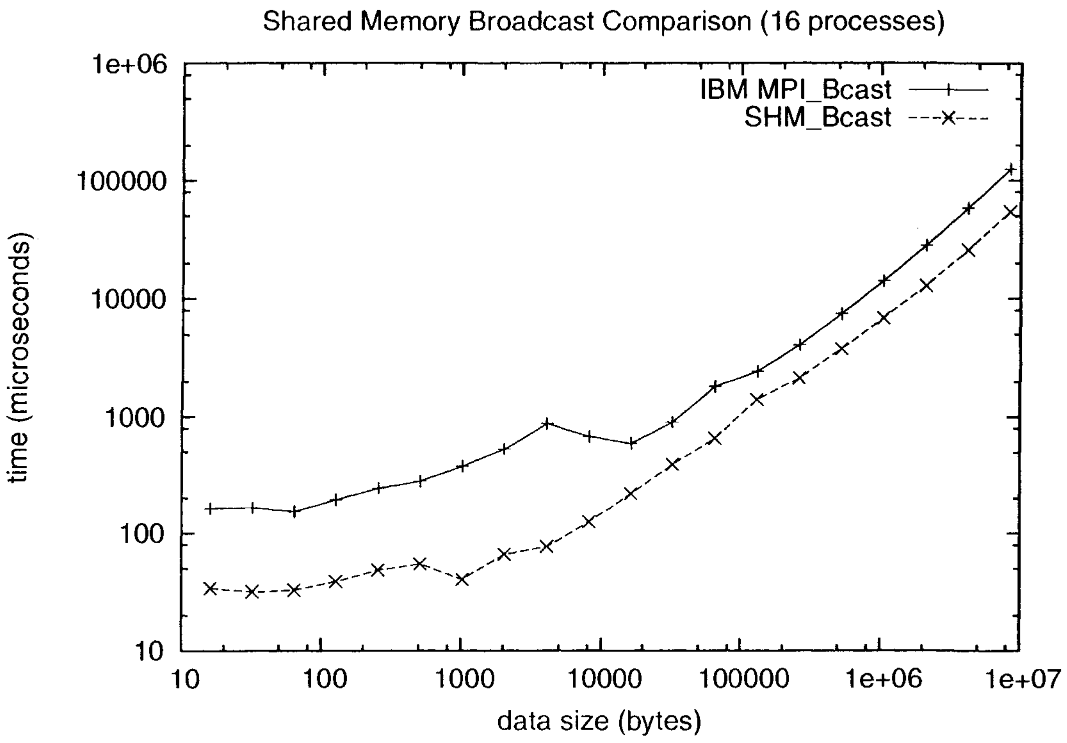


Figure 2.9 Performance of our broadcast implementation versus the vendor supplied broadcast on one SMP node

2.6 Performance Measurement

We use the following steps to measure the performance of a single operation:

(1) Assign values

(2) Purge cache

(3) Barrier call

(4) Start operation timing

(5) Execute collective operation

(6) End operation timing

(7) Verify values

(8) Allreduce operation to extract the node with the longest run time

Purging cache is necessary to ensure that data comes from main memory, not from cache.

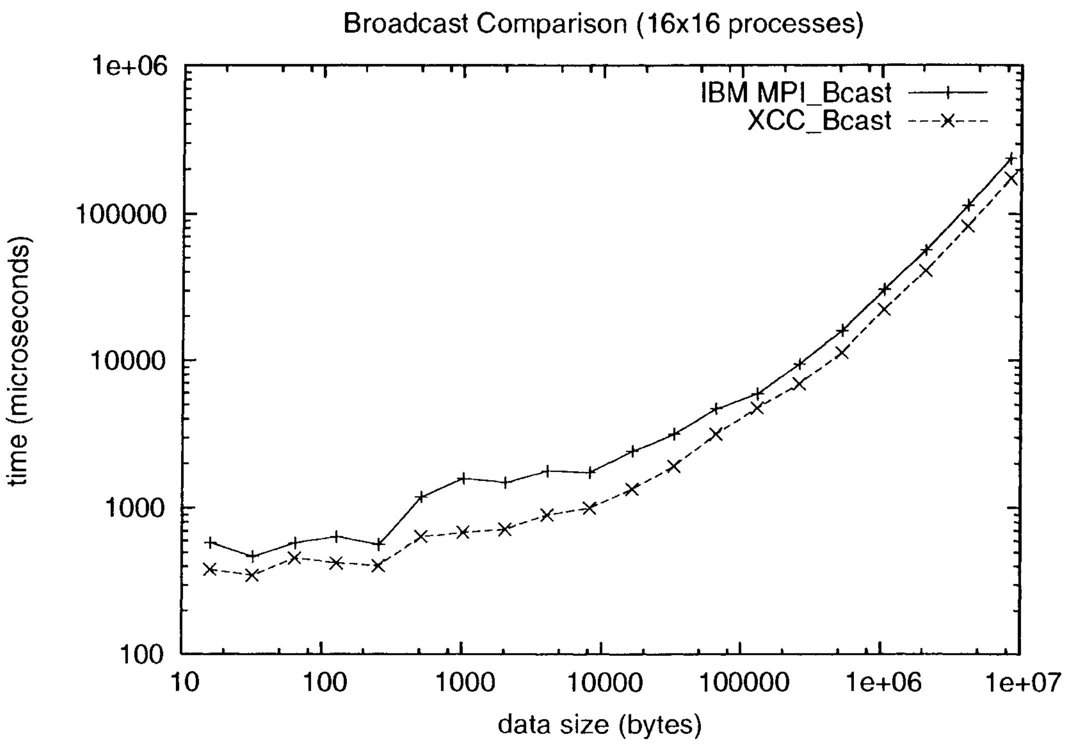


Figure 2.10 Performance of our broadcast implementation versus the vendor supplied broadcast on 16 SMP nodes

To purge cache, a memory block of L2 cache size is allocated; each byte in the block is then set to 0. This makes sure that no data used for communication is present in cache.

During our experiments we found that occasionally there are certain kind of "spikes" in our performance curve which can not be reproduced. We speculate it is due to the background system jobs. To eliminate this kind of "spike" and give more accurate results, our benchmarking procedure can set the parameters to determine how many runs we want for an experiment, and how many runs with "the longest run time" to be discarded. For example, if for an experiment we decide to run it 20 times (20 runs), we can discard the three runs with longest run time and compute the average of the rest seventeen runs. Our experimental results show that, when there is no irregular spike, this approach gives very close result with/without discarding runs. When there are irregular spikes, this approach can remove those irregular spikes and provides more accurate performance curves.

There are many collective communications in MPI; to make sure the implementations are

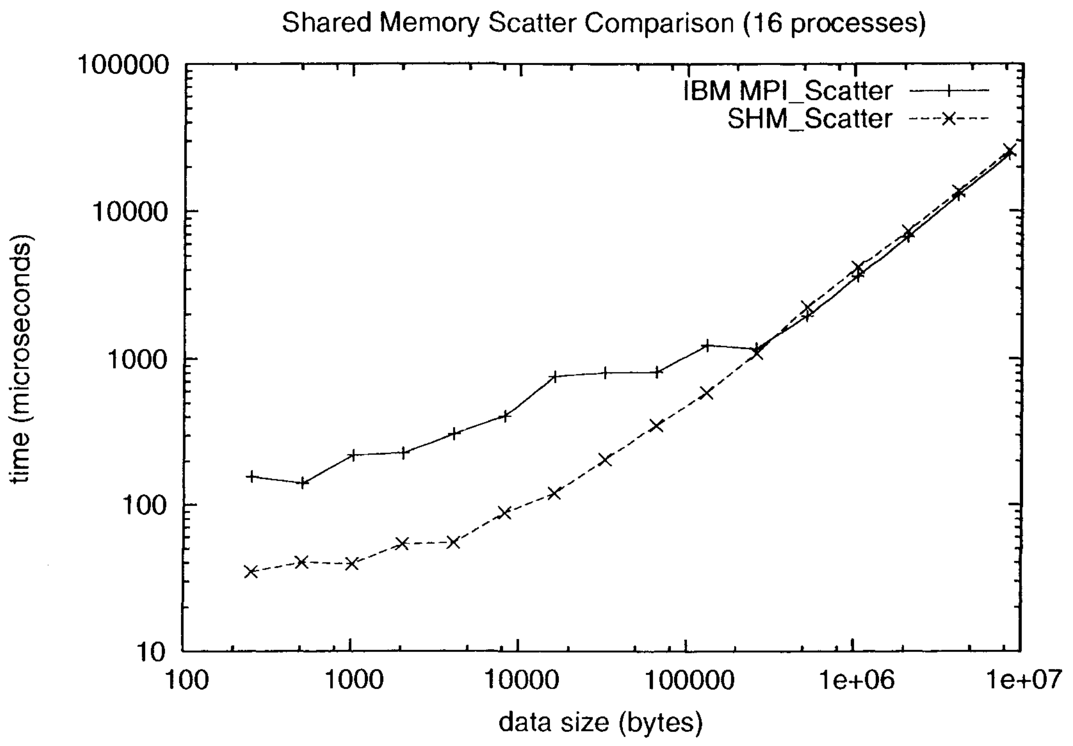


Figure 2.11 Performance of our scatter implementation versus the vendor supplied broadcast on one SMP node.

implemented correctly, we also designed routines to assign values to each message base on the type of collective communication and verify the correctness at the end of the operation. The verification routines can be set on or off by setting a parameter.

Each node starts timing after barrier synchronization. When the collective operation is finished, the node with the longest run time is identified. By doing this we can meet the condition "between the first process starting and the last process finishing the collective operation" as described by Worsch and co-workers [34]. The research efforts related to benchmarking MPI collective communications includes: Natawut and Lionel [37], Worsch and co-workers [34], SKaMPI [76], and the perftest of Gropp and Lusk for MPICH [35].

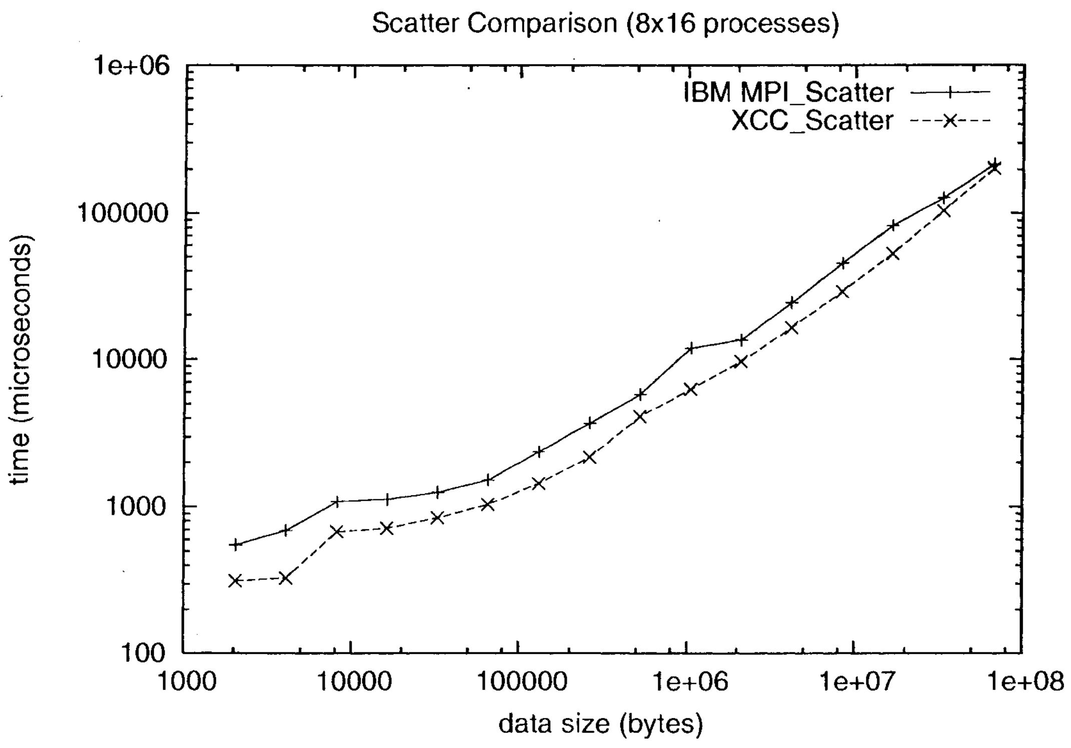


Figure 2.12 Performance of our scatter implementation versus the vendor supplied broadcast on 8 SMP nodes

2.7 Experimental Results

2.7.1 Broadcast

Figure 2.9 and 2.10 show the performance of our generic hierarchical implementation of broadcast against vendor's MPI_Bcast. Our approach is to select binomial tree algorithm for inter-node communication, and use shared memory operations for the intra-node layer communications. The result is usually at least 30% faster than MPI_Bcast, and sometimes more than 50% faster than the MPI_Bcast time, especially when the data size is small.

2.7.2 Scatter and Gather

Our selection for scatter is as follow: when message size is less than or equal to 128K, use shared memory operations; if message is larger than 128K, switch to flat tree algorithm that implemented using vendor's send/receive implementations. The results in Figures 2.11

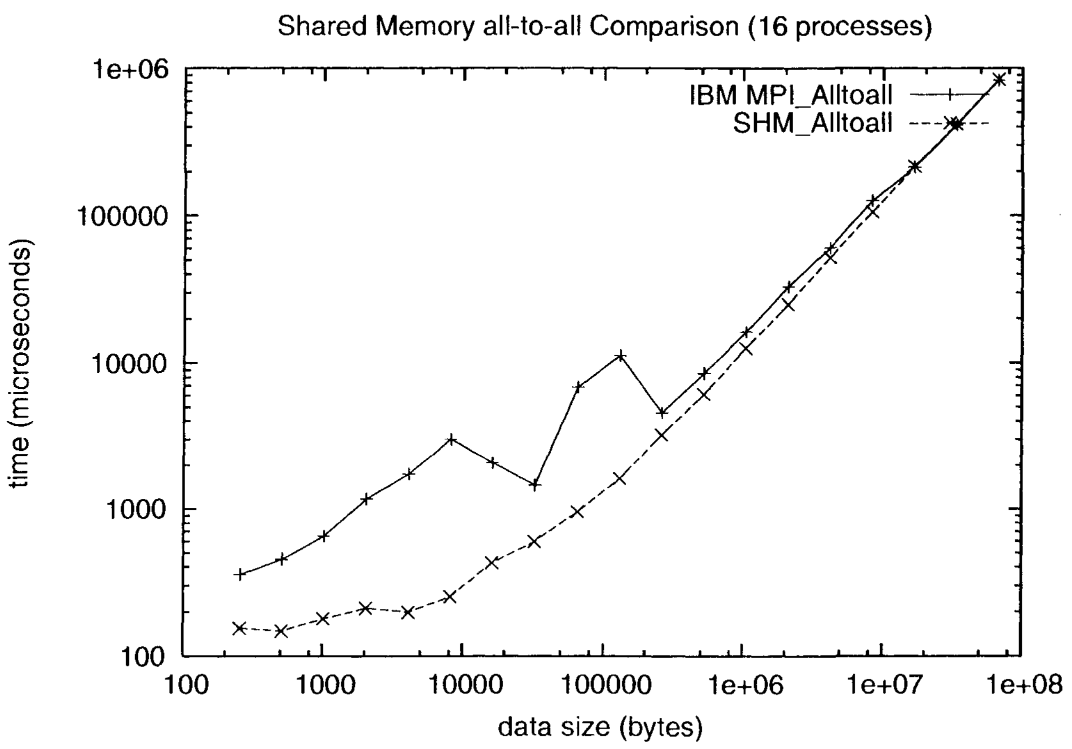


Figure 2.13 Performance of our all-to-all implementation versus the vendor supplied broadcast on one SMP node

and 2.12 demonstrate that when the total data size is less than 128K, the run time of the concurrent group access is just about 40% faster than MPI_Scatter on a single node with 16 MPI tasks. When using several nodes with 8 MPI tasks per node, our implementation can be as much as 30% faster than MPI_Scatter. Our shared-memory implementation performs better than the vendor implementation from 8K to 128K message size due to less synchronizations are required for shared memory operations at this range. Gather is just the inverse of scatter and the performance with the appropriate operations is similar.

2.7.3 All-to-all

Figure 2.13 shows the performance comparison of two shared-memory all-to-all approaches. Within an SMP node, the shared-memory all-to-all algorithm performs better than the other approaches when the data size is small to medium. However, our approach cannot perform better than the pair-wise exchange on the testing platform even when data size is small. When

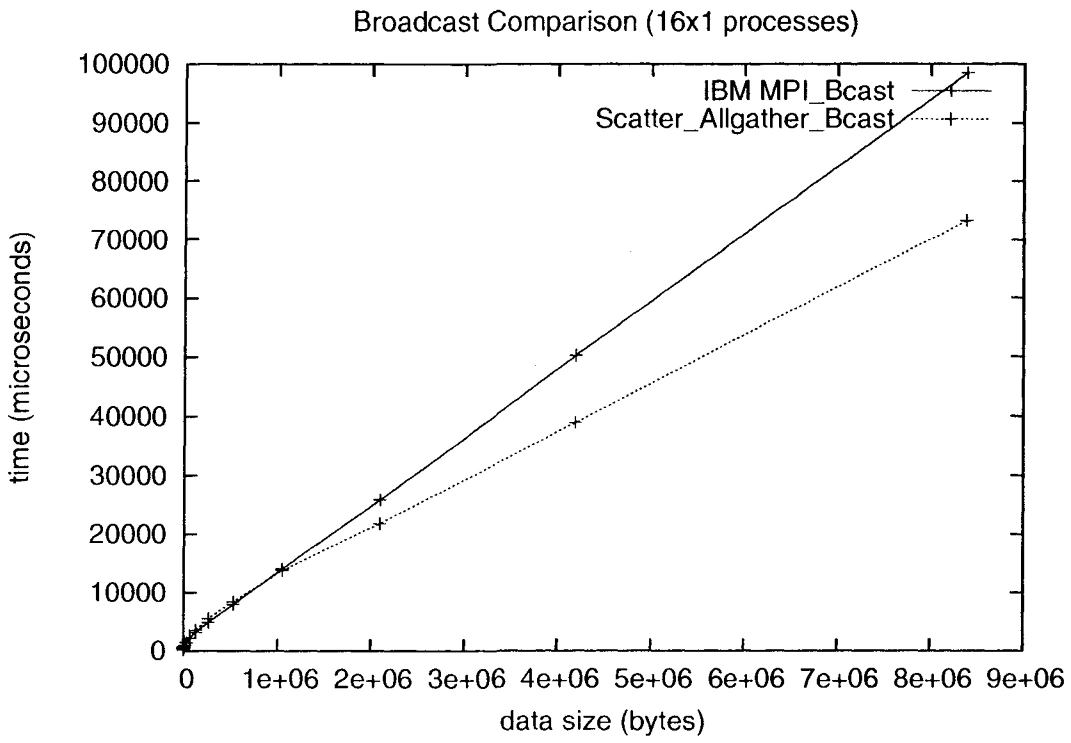


Figure 2.14 Performance of two stage broadcast of MPICH implementation versus the vendor supplied broadcast and our modified two stage broadcast versus the vendor supplied broadcast on 16 SMP nodes

we are not using the pair-wise exchange algorithm and assuming only one process in each node is in charge of communication with other nodes, we have to do a shared-memory gather, inter-node all-to-all, and a shared-memory scatter set of steps. Even when the data size is small, group_comm has to re-arrange data so that shared-memory gather and scatter can be done in one stage; otherwise it is B stages of the shared-memory scatter or gather. This extra cost of memory operations are larger than the performance gain of the shared-memory all-to-all algorithm. Our choice here is to use pair-wise exchange when all-to-all operation involves inter-node communications, and use shared memory operations when only one SMP node is used.

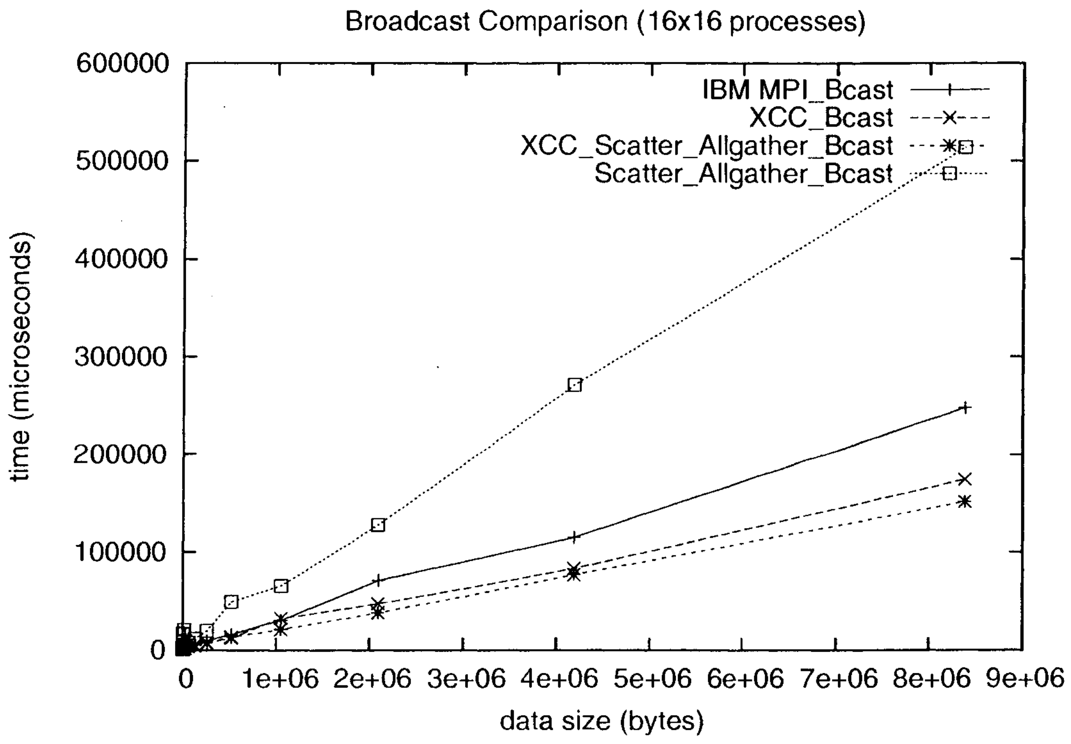


Figure 2.15 Performance of two stage broadcast of MPICH implementation versus the vendor supplied broadcast and our modified two stage broadcast versus the vendor supplied broadcast on 16 SMP nodes

2.7.4 Two Stage Algorithms

Barnett *et al.* discussed broadcast algorithm [21] uses two stages: first message segments are scattered to all processes, then each process collects data by Allgather. The algorithm works well with large data sizes and is implemented in MPICH 1.2.5. Figure 2.14 shows the performance of the MPICH implementations of this algorithm compared to the IBM MPI_Bcast on large data sizes when communication involves only inter-node messages. However, if intra-node communication is optimized only with send/receive, the algorithm performance degrades due to extra memory copies. Figure 2.15 shows the same algorithm run on 16 nodes with 16 MPI tasks; the performance of this algorithm is much worse than IBMs MPI_Bcast. We modified the algorithm so that it works only on the inter-node level. On the intra-node level we use shared-memory operations which incur no extra memory copy, and this implemen-

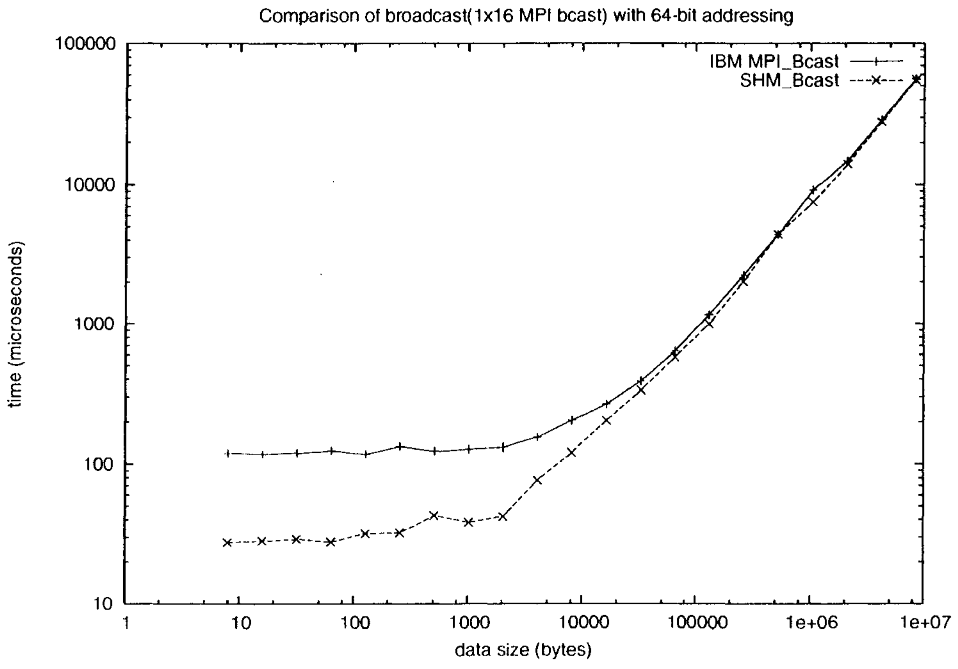


Figure 2.16 Performance of our broadcast implementation versus the vendor supplied broadcast on one SMP node, with 64-bit addressing

tation even outperforms our broadcast implementation using a binomial tree for inter-node communication.

2.7.5 A Performance Update

The experiments in this chapter were conducted in 2003 and the results were published in the Proceeding of the IASTED International Conference on Parallel and Distributed Computing and Networks conference (PDCN) in February 2004. In October 2004 the vendor upgraded the MPI implementation on the IBM SP cluster, with the major improvement on collective communications that can take advantage of the shared memory architecture. The memory copy operation within an SMP node is also greatly improved so the extra-memory copies problem that causes a sequential algorithm to perform better than a binomial tree algorithm is corrected. The code must be compiled with 64-bit addressing to utilize shared memory for collective operations, otherwise it will still use shared memory send and receive for collective communications.

We have compared our shared memory broadcast implementation with IBM's new 64-bit shared memory broadcast. Without 64-bit addressing support, the performance comparison is similar to 2.9. With 64-bit addressing support, our generic implementation still outperform vendor's implementation for small message size up to 16K; after message size is larger than 16K the performance are very close and our implementation is slightly better than the vendor's implementations most of the time. Figure 2.16 shows the result.

2.8 Summary

To achieve optimal communication performance on a cluster of SMP nodes, not only do we need optimal algorithms for the inter-node communications, but also good shared memory communication schemes that can be adjusted according to different collective communication characteristics. By decomposing collective communications into shared-memory operations, we provide an approach to design and improve collective communications within an SMP node. Our experimental results show that our approach can utilize SMP clusters better than the vendor's implementations. For the rest of this dissertation, if an intra-node collective communication is composed using concurrent memory access features, we call it shared memory collective communication. The approaches described in this chapter focus on designing shared memory collective communications, without any overlapping between inter-node/intra-node communications. To further utilize the SMP architecture, we discuss how to develop generic overlapping mechanisms in the next chapter.

CHAPTER 3. OVERLAPPING INTER-NODE/INTRA-NODE COLLECTIVE COMMUNICATIONS

3.1 Introduction

The second feature on SMP clusters that can be used to improve the performance of collective operations is to overlap inter-node and intra-node collective communications.

One approach to overlap inter-node and intra-node communications is to map MagPIe's WAN/LAN communications model onto inter-node/intra-node communications layers. Their approach assumes that there is a big difference of latency/overhead between different communication layers (on the WAN environments, the difference is at least two order of magnitude). A communication tree that overlaps inter-node/intra-node communications can be calculated base on this difference. However, on SMP clusters the latency and overhead between the inter-node/intra-node layers does not have such a huge difference (on our testing platforms, the difference of overhead is less than 50%); applying MagPIe's approaches on SMP clusters can only have very limited performance improvement.

Another approach was proposed by Tipparaju *et al.* [9]. In their approach they used remote direct memory access (RDMA) to implement pipelined version of inter-node collective communications, overlapping with shared memory collective communications. The RDMA functions were provided by LAPI, which is IBM's propriety low level communication library. At the time this dissertation is written there is no portable LAPI available, thus we consider Tipparaju's method is a platform specific approach. However, a new MPI implementation that is still under development, OpenMPI [73], may include a version of portable LAPI into its communication library and Tipparaju's approach may become portable in the future.

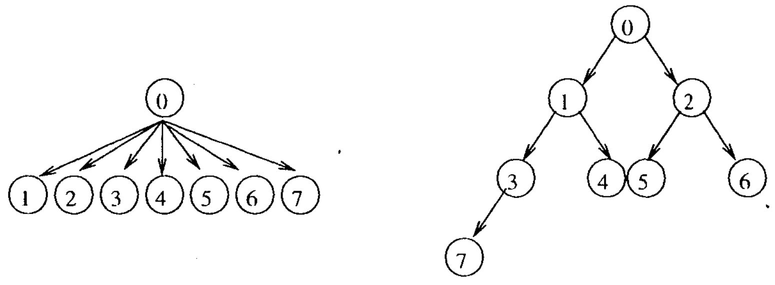
In this chapter we first explore several design issues of collective operations on the inter-node layer, then we discuss why RDMA functionalities in MPI-2 standard and ARMCI are not suitable for a generic approach to overlap inter-node/intra-node communications, followed by our generic mechanism to overlap inter-node and shared memory collective communications. A generic programming model for designing collective communications on SMP clusters is constructed based on this generic overlapping mechanism and the shared memory collective communications described in the previous chapter. Several collective communications are developed based on this model, and the performance results and comparison with the other implementations are also shown in this chapter.

3.2 Algorithms for the Inter-node Collective Communication

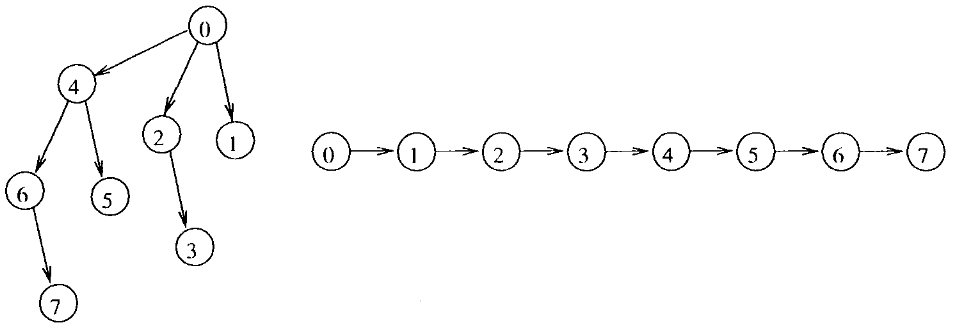
Before discussing the overlapping mechanisms of inter-node/intra-node communications, this section gives a brief introduction of the algorithms for the inter-node layer collective communications. Different algorithms for collective communications on the inter-node layer have been developed during the last two decades [20, 21, 22, 24, 23, 15, 16, 17, 25, 26, 27]. Their assumption is that there is no hierarchical structure in communications, and the underlying network is fully connected. These algorithms can be roughly classified into tree structures as used by ACCT [14], and they are shown in Figure 3.1.

The four major tree algorithms for collective communications are flat tree (sequential tree), chain tree, binary tree and binomial tree. With a flat tree algorithm, the root node sends messages to all the other nodes one by one in the tree, as shown in Figure 3.1(a). Using a chain tree algorithm, the root node sends a message to its child node, then the child node forwards the message to it's child node and so on until the last node in the tree receives the message. This is shown in Figure 3.1(d). Ring algorithm, which is not shown in Figure 3.1, is a variation of chain tree algorithm in which the last node also sends messages to the first (root) node.

When using a binary tree algorithm, a parent node sends messages to its two child nodes. The child nodes then send messages to their two child nodes until every node in the tree receives its message. With a binomial tree algorithm, the root node (rank 0) first sends a message to a



(a) Flat tree (b) Binary tree



(c) Binominal tree (d) Chain tree

Figure 3.1 Tree structures for collective communications.

node with rank $p/2$ (p is the number of processes in the communication group, here we assume p is a power of two), then the communication group is divided into two sub-groups; one group contains the original root node and the root node again sends a message to the node of rank $p/4$, and another group with node rank $p/2$ as the root node, and this new root node sends a message to the node with rank $p/2 + p/4$. The process is repeated recursively until every node in the tree receives its message.

Most collective communications can be developed base on the four tree algorithms; the two stages broadcast is also a composition of these algorithms. These algorithms certainly are not the only algorithms for collective communications; the other algorithms such as pair-wise exchange, recursive doubling are also commonly used for certain collective operations. We use these four algorithms in this dissertation due to their popularity and easy for understanding. The methods we developed in this dissertation can be applied on these algorithms and the performance improvement can be achieved on different SMP clusters.

3.3 The Programming Model

Figure 3.2 outlines the programming model to design collective communications on SMP clusters. There are three layers in this model: the top layer is the inter-node communication layer, the bottom layer is the intra-node communication layer, and the middle layer is the overlapping mechanisms.

There are different methods to implement collective communications on the inter-node layer and the intra-node layer. Base on the methods used on the inter-node/intra-node layers we may have different choices of overlapping mechanisms. We listed all possible methods on each layer and all possible combinations to design collective communications on SMP clusters in a generic approach. The existing generic approaches which are outlined by dash lines means the approaches taken by the other portable MPI implementations, and can be used across different SMP clusters. We add several new generic approaches that are outlined in solid lines. The details of Figure 3.2 are discussed in the following subsections.

3.3.1 The Inter-node Communication Layer

Most MPI implementations, such as MPICH, implement one or several of the algorithms described earlier with blocking sends and receives. A blocking implementation usually incurs less software overhead than a non-blocking implementation, thus performs well when the communication is latency bound (usually small messages) [39]. However, when a communication is bandwidth bound [39] (usually large messages), the blocking implementation may not be the best choice. Consider using chain tree algorithm to broadcast a large message with a blocking implementation. Non-root nodes can not start processing the message until the whole message is received. This can lead to very poor performance.

To improve the performance of sending large messages, pipelining is taken to implement inter-node collective communications. Using pipelining [10, 14], a message is first broken into many segments. The sender then sends the message segment by segment instead of the whole message, thus non-root nodes can process the message once a segment is received. This is shown in Figure 3.3. We usually use non-blocking sends and receives to implement the

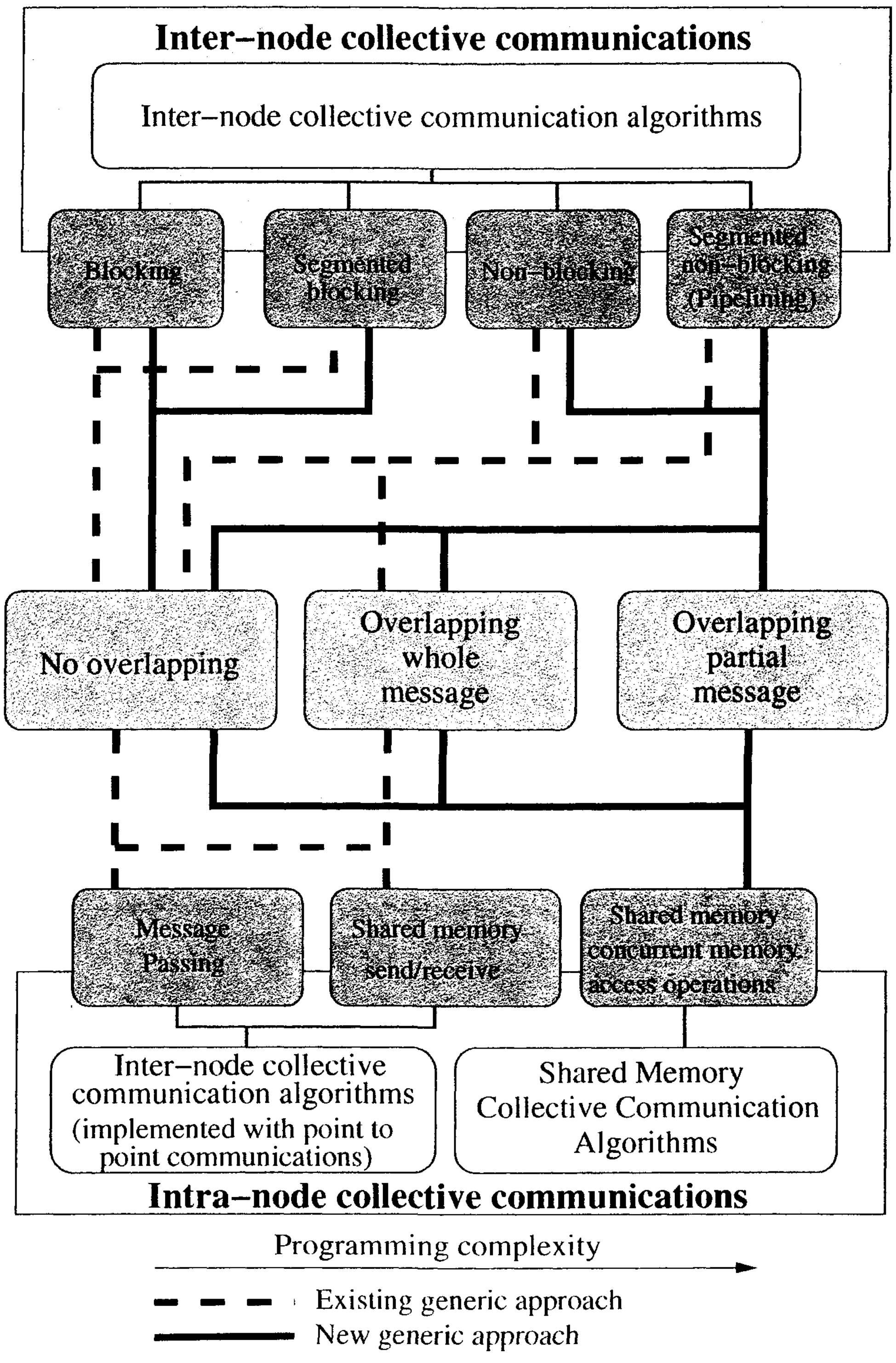

Inter–node collective communications
Inter–node collective communication algorithms
Blocking
Segmented blocking
Non–blocking
Segmented non–blocking (Pipelining)
No overlapping
Overlapping whole message
Overlapping partial message
Message Passing
Shared memory send/receive
Shared memory concurrent memory access operations
Inter–node collective communication algorithms (implemented with point to point communications)
Shared Memory Collective Communication Algorithms
Intra–node collective communications
Programming complexity
Existing generic approach
New generic approach

Figure 3.2 The programming model for designing collective communications on SMP clusters.

pipelined version of an algorithm. Since a blocking implementation blocks the sending or receiving processes when processing a collective communication, we can only use the non-blocking implementations to design overlapping mechanisms for overlapping inter-node/intra-node communications.

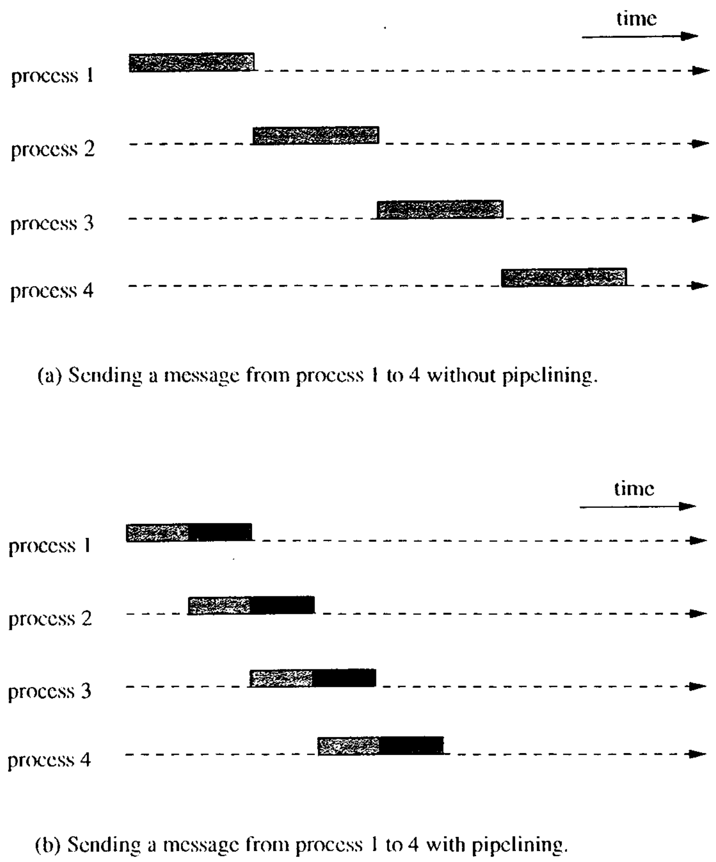

time
process 1
process 2
process 3
process 4
(a) Sending a message from process 1 to 4 without pipelining.
time
process 1
process 2
process 3
process 4
(b) Sending a message from process 1 to 4 with pipelining.

Figure 3.3 Processing a message, without pipelining and with pipelining.

An algorithm can be implemented as a whole message or segmented, either using MPI blocking or non-blocking send/receive, thus we have four different implementations for a collective communication algorithm on the inter-node layer. This is shown as the top layer in Figure 3.2.

The segmented blocking implementations may also improve the performance through pipelining; however, they block the calling processes and make it not possible for overlapping. The non-blocking whole message implementation does not take advantage of pipelining. Both approaches seldom provide good performance; we list them in the model for completeness.

Segmented non-blocking (pipelined) implementations incur more software overhead, require careful calculation for message segments, making them difficult to implement and even more difficult to tune to achieve good performance. However, those are the implementations that can utilize pipelining and overlapping. With proper tuning, a pipelined implementation usually gives the best performance when the communication is bandwidth bound. Some research projects, such as ACCT [14, 13] or MagPIe [11, 10], use this approach to design their inter-node collective communication libraries.

When implementing our generic, collective communications library, we use only MPI_Send and MPI_Recv for blocking implementations, and use MPI_Isend and MPI_Irecv for non-blocking implementations. We did not encounter any problems in porting our implementations to different platforms and overall performance was not impacted by this choice. Some approaches used MPI_Rsend to implement collective communications, we consider it not realistic to be used in real problems. MPI_Rsend assumes that the corresponding receive is already posted. For a scientific application that is using collective communications, the assumption that a certain process finishes its computation stage earlier than another process can be too restrictive.

3.3.2 The Intra-node Communication Layer

The existing generic approaches on this layer, as mentioned in the previous chapter, are to use the point-to-point based collective communication algorithms for the inter-node layer. Point-to-point communications can be through the communication network, or through shared memory send/receive. With the shared memory collective communications we developed in the previous chapter, there are three generic methods on this layer. We have discussed the detail of each approach in the previous chapter. In Figure 3.2, the bottom layer shows the intra-node layer communications; each block represents a method on this layer.

3.3.3 Inter-node/Intra-node Overlapping Mechanisms

Most collective communication algorithms developed during the last decades focus on the top layer with the assumption that each node has one processor. Golebiewski *et al.* [18] and Kielmann *et al.* [11, 10] developed collective communication algorithms for two communication layers by utilizing the difference of communication latency between two layers. Their algorithms are implemented with point to point communications on both layers, without overlapping. MagPIe provides an overlapping mechanism on the inter-node/intra-node layer, but the overlapping mechanism is also based on point-to-point communications.

The approach by Tipparaju *et al.* [9] that uses RDMA functions provided by LAPI, as mentioned earlier, is a platform specific approach for IBM platforms. Since our final goal is an automatic tuning system for different SMP clusters, our first idea was to replace the platform specific RDMA provided by LAPI with existing generic RDMA libraries.

3.3.3.1 RDMA Functions in Other Libraries.

There are two libraries that provide generic RDMA functions on different platforms: MPI-2 [67] and the Aggregate Remote Memory Copy Interface (ARMCI) [80, 81]. The MPI-2 standard was first released in 1996, and only recently have the RDMA features been regarded as mature. ARMCI provides similar RDMA functions, but with much simpler rules than the complex rules set by MPI-2 standard.

At the moment, the two generic RDMA approaches pose difficulties for using them to implement generic inter-node layer collective communications. The major reason is that a memory segment must first be registered to be used by RDMA functions. Data in a memory segment which is allocated by *malloc()* or *calloc()* must be copied to a registered memory segment to be accessible to RDMA functions. This means that if we implement non-blocking collective communications using these generic RDMA functions, we would need one extra copy on both the sender and receiver sides. This could lead to performance degradation. Requiring an end-user to allocate memory using the functions provided by these RDMA libraries would be asking them to go through their entire MPI application and make changes accordingly. This is both unrealistic and error-prone. For these reasons, we have chosen not to use RDMA to implement mechanisms for overlapping inter-node/intra-node communications.

3.3.3.2 The Overlapping Mechanisms

Another approach is to modify existing non-blocking segmented implementations on the inter-node layer to overlap inter-node/intra-node communications. The idea is very straightforward: When processing a collective communication, the group coordinator posts non-blocking sends for a message and then starts shared memory collective communications. When a non-blocking receive is posted, shared memory collective operations can be started as soon as a message segment is received. At the same time the communication layer can continue receiving the other message segments. In other words, we can treat shared memory collective communications as computation.

The major difficulty hidden behind this seemly simple strategy is that when using MPI non-blocking calls we cannot merely overlap the entire message segment as in Tipparaju's approach. Let $g(m)$ be the gate value, and $os(m)$ be the send overhead of sending a message of size m as defined in parameterized LogP model [11, 10]. When using non-blocking sends to send a segment, the next segment of size m can not be sent before $g(m)$. The overhead $os(m)$ is usually smaller than $g(m)$, thus the best theoretical interval for overlapping is $g(m)$ - $os(m)$. When we start shared memory collective operations after posting a non-blocking send, it is very possible that the cost of the shared memory collective communications of size m is larger than $g(m)$ - $os(m)$. Simply overlapping a whole segment may delay sending the next message segment. Therefore, when using MPI non-blocking functions to overlap inter-node/intra-node communications we also have to consider overlapping only partial message segment. On the other hand, when communications are latency bound, using non-blocking segmented implementations can only make the performance worse since overhead is higher: thus the best approach may be to not overlap at all.

This lead us to construct the programming model outlined in Figure 3.2. The dashed lines indicate existing generic approaches to design collective communications on SMP clusters.

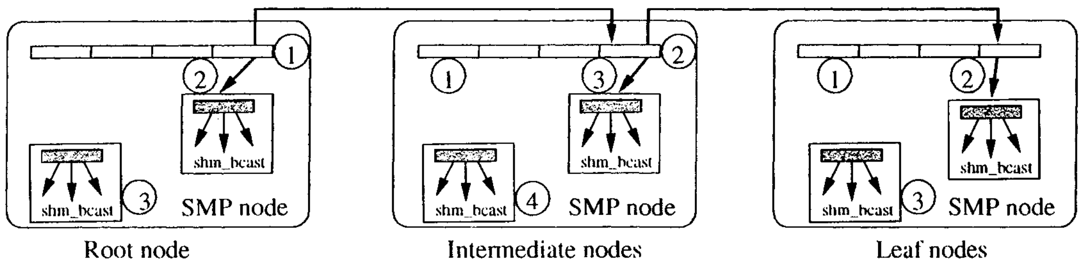


Figure 3.4 Overlapping inter-node/intra-node communications for broadcast

The solid lines are new generic approaches we added. For example, blocking implementations (either whole message or segmented) cannot be combined with any overlapping mechanism, but they can still take advantage of using shared memory collective communications on the intra-node layer to improve overall performance.

The programming complexity of implementing collective communications in Figure 3.2 is roughly increasing from left to right (with the exception of non-blocking and segmented blocking on the inter-node layer). For a dashed line in Figure 3.2, inter-node layer collective communications can be used directly on the intra-node layer. For a solid line, new algorithms or implementations are required for good performance.

3.4 Implementations of Collective Communications and Experimental Results

The four most commonly used algorithms for inter-node collective communication are described in an earlier section. Each has strengths and weaknesses under different circumstances; some may even be entirely unsuitable for a given collective operation. In this section we discuss our optimization of some collective operations on SMP clusters base on the programming model outlined in Figure 3.2. Our focus is on how we design overlapping versions of these collective communications.

3.4.1 Broadcast

Most existing MPI implementations of broadcast use a blocking implementation of a binomial tree algorithm on both layers without overlapping communication at different levels. ACCT uses segmented non-blocking implementation of different algorithms, but only for the inter-node layer. MagPIe measures the difference of the gate value and overhead between two communication layers, and derives a communication tree accordingly. In their overlapping mechanisms, an inter-node point-to-point communication overlaps an intra-node point-to-point communication, so we regard it as a generic overlapping mechanism for a whole message segment.

Our broadcast implementations cover almost every path in the programming model in Figure 4.5, which covers most of the existing approaches. Our new generic approach is to overlap inter-node communications with shared memory collective communications.

Figure 3.4 outlines the mechanisms to overlap inter-node/intra-node communications for broadcast. The root node (1) posts a non-blocking send(s) of a segment, starts (2) the shared memory broadcast, and then repeats (1) and (2) until the last segment is processed. The last step (3) is executed if the shared memory broadcast segment size is smaller than the segment size for the inter-node layer broadcast. Intermediate nodes initially (1) post all non-blocking receives. Once a segment is received, it (2) posts a non-blocking send for this segment and then (3) starts a shared memory broadcast. The intermediate nodes repeat (2) and (3) until the last segment is processed and then finish the operation (4) if another shared memory broadcast is required. Leaf nodes operate in the same sequence as intermediate nodes, but without forwarding message segments: thus they (1) post non-blocking receives, (2) wait for a segment to be received and then start the shared memory broadcast, repeating (1) and (2) until the last segment is processed, and (3) process the last shared memory broadcast if necessary. If a node has multiple child nodes (such as in a binomial or binary tree broadcast), it will post all non-blocking sends to all child nodes before starting the shared memory broadcast.

Figure 3.5, 3.6, and 3.7 show the results of different broadcast implementations with an 8 MB message on 3 different platforms. Implementation 1 is a blocking implementation on both

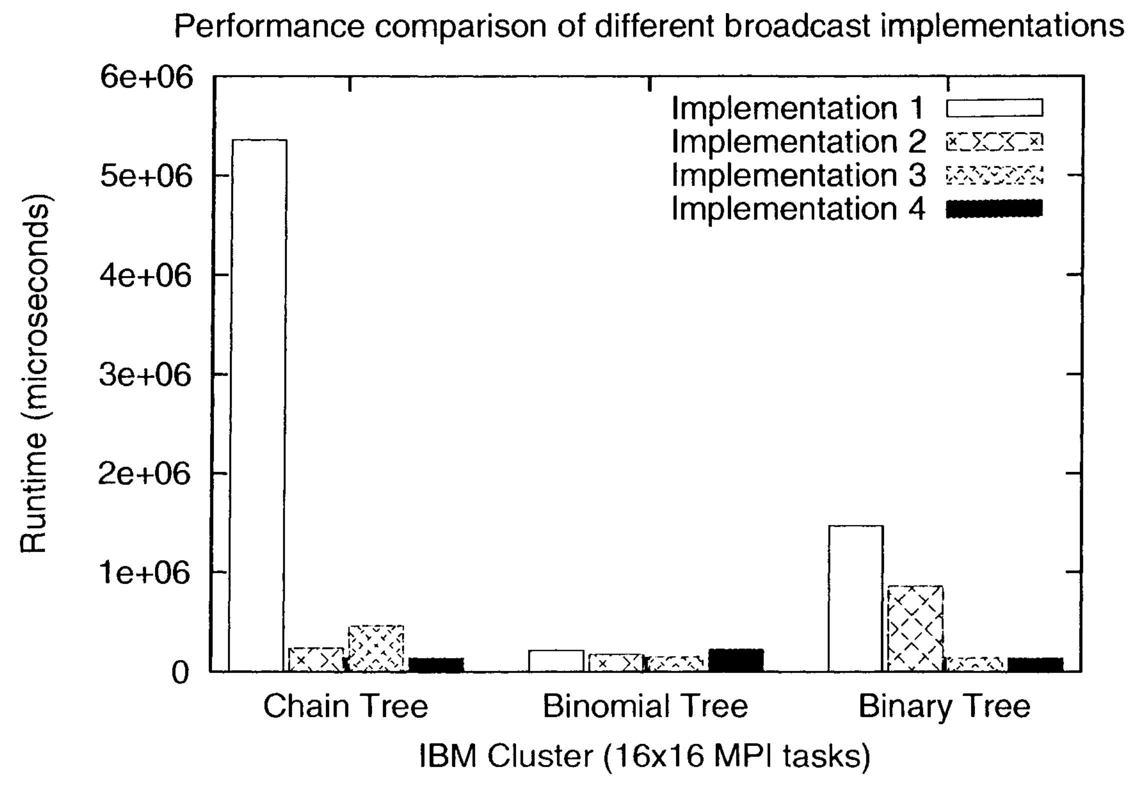


Figure 3.5 Performance comparison of different broadcast implementations on the IBM Cluster, using 8MB messages

layers (most MPI broadcast uses this approach). Implementation 2 is a non-blocking segmented implementation on both layers (the same approach as ACCT and MagPIe). Implementation 3 uses the blocking implementation on the inter-node layer and a shared memory broadcast on the intra-node layer (the same as in our previous work [113]). Implementation 4 uses the segmented non-blocking implementation on the inter-node layer and overlaps with the shared memory broadcast on the intra-node layer. In short, we can look at implementations 1 and 2 as existing generic approaches, while implementations 3 and 4 represent new generic approaches in the programming model.

From Figure 3.5, 3.6, and 3.7 we can see that layers of optimization have different effects on different algorithms. Implementation 2 greatly improves performance on all platforms for the chain tree algorithm. Implementation 3 provides good performance improvement only when the number of processes per node is large enough such as on IBM SP (16 processors per node).

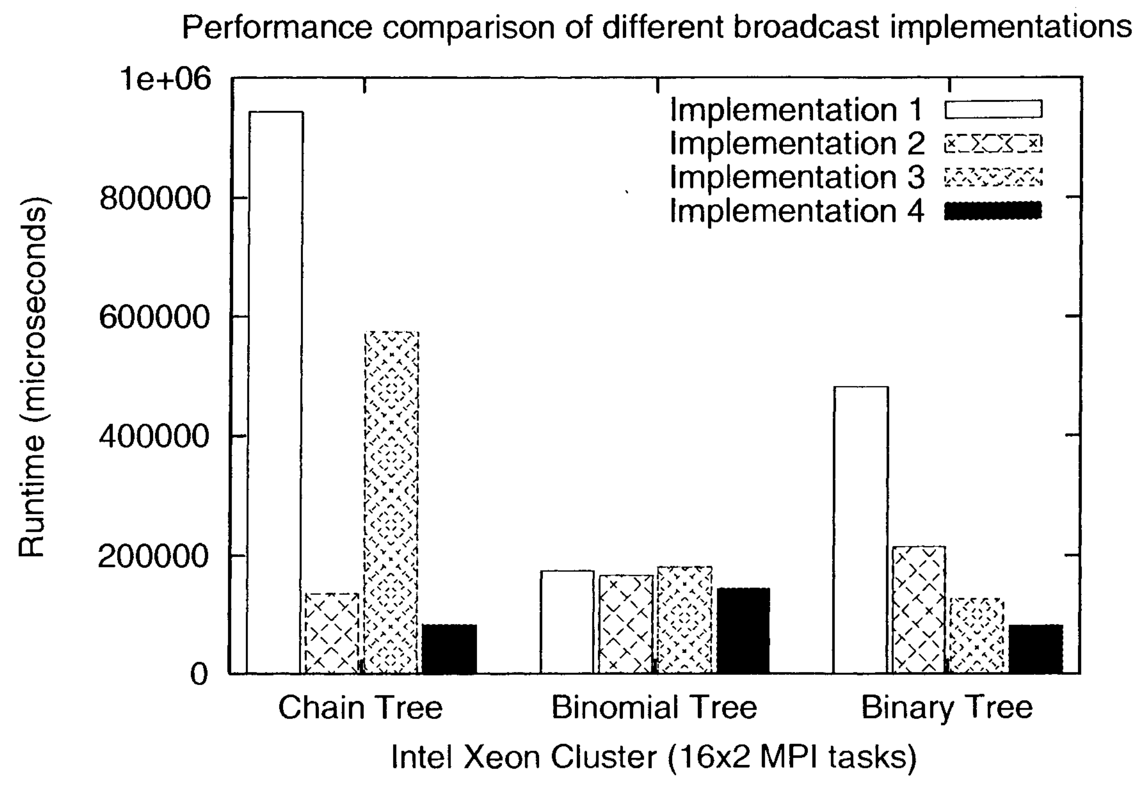


Figure 3.6 Performance comparison of different broadcast implementations on the Intel Xeon Cluster, using 8MB messages

Implementation 4 provides the best performance across all three platforms. Overlapping for the chain tree broadcast can hide almost all the cost of shared memory broadcast on the IBM SP.

For the binary tree algorithm, each layer of optimization provides a certain degree of performance improvement. Implementation 4 provides the best performance on the IBM SP and the Intel cluster. In fact, the overlapping version of the binary tree and chain tree algorithms provide similar performance on all three platforms.

As for binomial tree algorithm, although implementation 1 provides better performance than the blocking version of the other two algorithms, different optimizations show only very limited performance improvement and sometimes even performance degradation. If our optimization stops at implementation 2 as in existing generic optimizations, the binomial tree still performs better than the other algorithms. However, when we optimize to implementation 4,

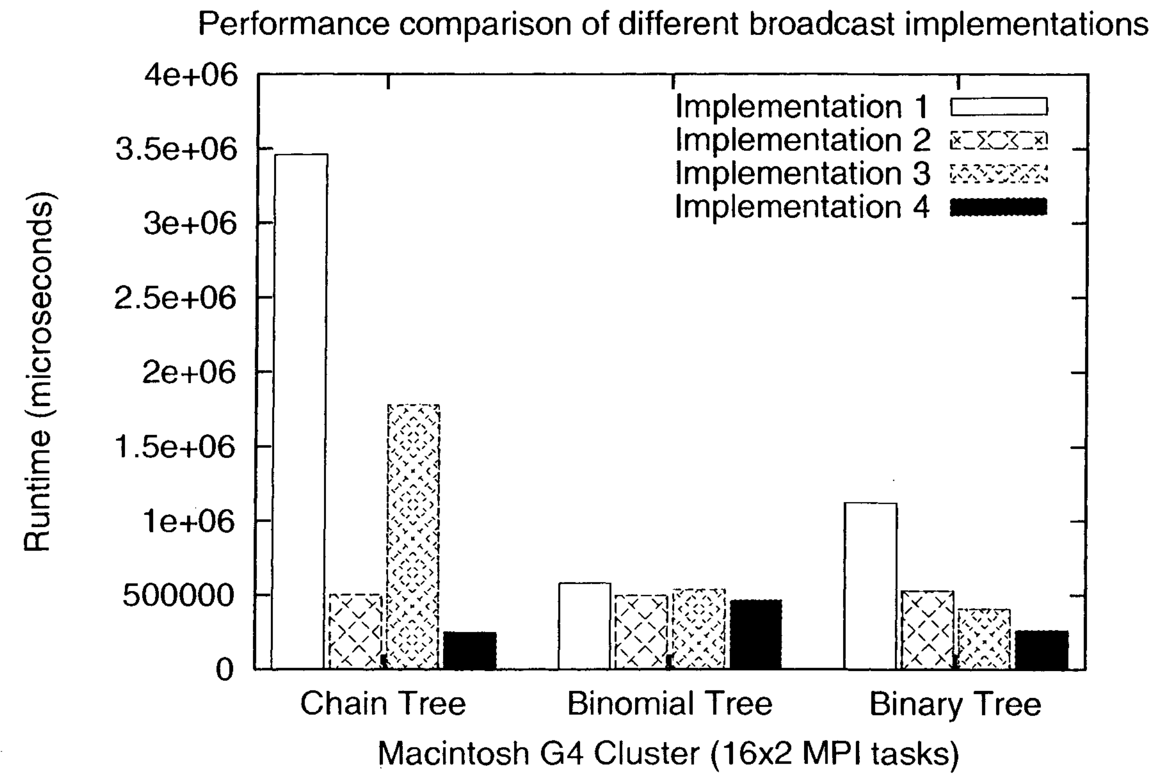


Figure 3.7 Performance comparison of different broadcast implementations on the G4 Cluster, using 8MB messages

our binomial tree implementation always performs worse than the other two algorithms.

Figure 3.8, 3.9, and 3.10 show the best results selected from our implementations against MPI_Bcast on each platform. Depending upon the message size, the performance improvement is 20% to 46% on the IBM SP; 27% to 63% on the Intel cluster, and 27% to 51% on the G4 cluster. Table 3.1 shows the details of which implementation is selected for a given message size on the three platforms. Except for message sizes less than 8KB on the G4 cluster and between 1KB and 8KB on the Intel cluster, our new generic optimizations show better performance than the existing approaches. Worth noting is that implementation 4 of both the chain and the binary tree algorithms performs much better than the two stage broadcast (scatter, all-gather) used by MPICH for broadcasting large messages.

The performance of an implementation is a function of {*algorithm, number of nodes, number of processes/processors per node, message size, segment size, and overlapping size*}; to the best

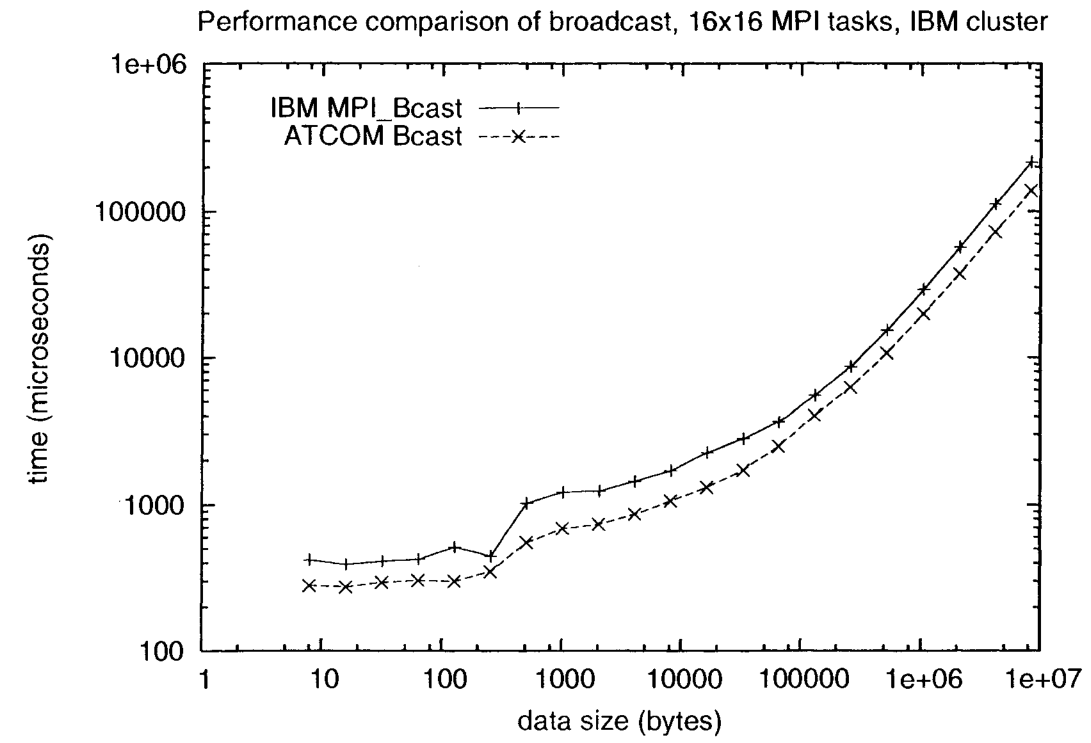


Figure 3.8 Performance comparison of broadcast on the IBM Cluster

of our knowledge there is no such performance model that covers inter-node communications with shared memory collective communications. A new performance model is needed to explain the effects of layers of optimization on different algorithms, and we will discuss this new performance model in the next chapter.

3.4.2 Scatter and Gather

Not all collective operations can take advantage of overlapping inter-node/intra-node communications. When the amount of data for inter-node communications is substantially larger than the intra-node communication data, the performance improvement can be very small. Consider scattering a message of 8M to 4x4 processes: the possible overlapping data size is only 1/15 of the intra-node communication data size. The cost of an intra-node scatter of 512K is almost negligible compared to the cost of sending out 7.5M of data to the inter-node

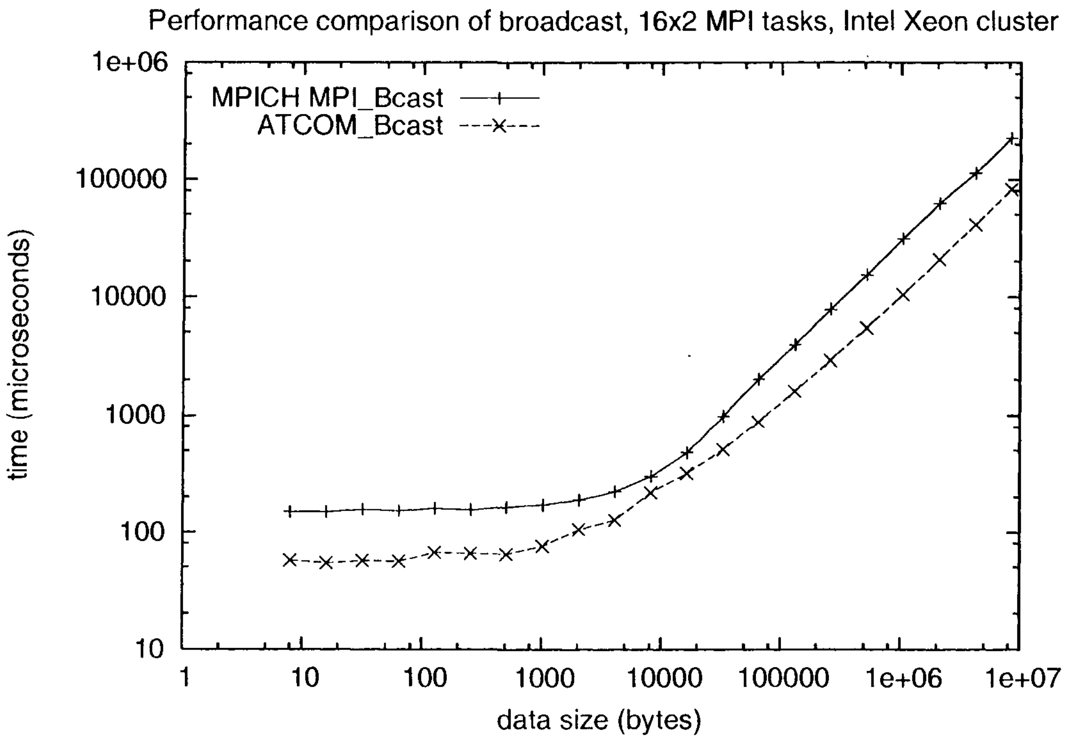


Figure 3.9 Performance comparison of broadcast on the Intel Xeon Cluster

layer. For this reason we did not implement the overlapping version of scatter or gather. However, for scattering or gathering small messages, we can still use concurrent memory access for intra-node scatter or gather to improve the performance since latency can be reduced. The best strategy is to use the blocking implementation for inter-node scatter and shared memory operations for intra-node scatter. We have shown the performance enhancement in one of our work [113] and in the previous chapter.

3.4.3 All-gather

It is not realistic to implement a collective communication following every path in the programming model in Figure 3.2. The reason we implemented broadcast following most paths in the programming model is that the results of broadcast can provide basic performance metrics for the other collective communications and give us insight into different algorithms.

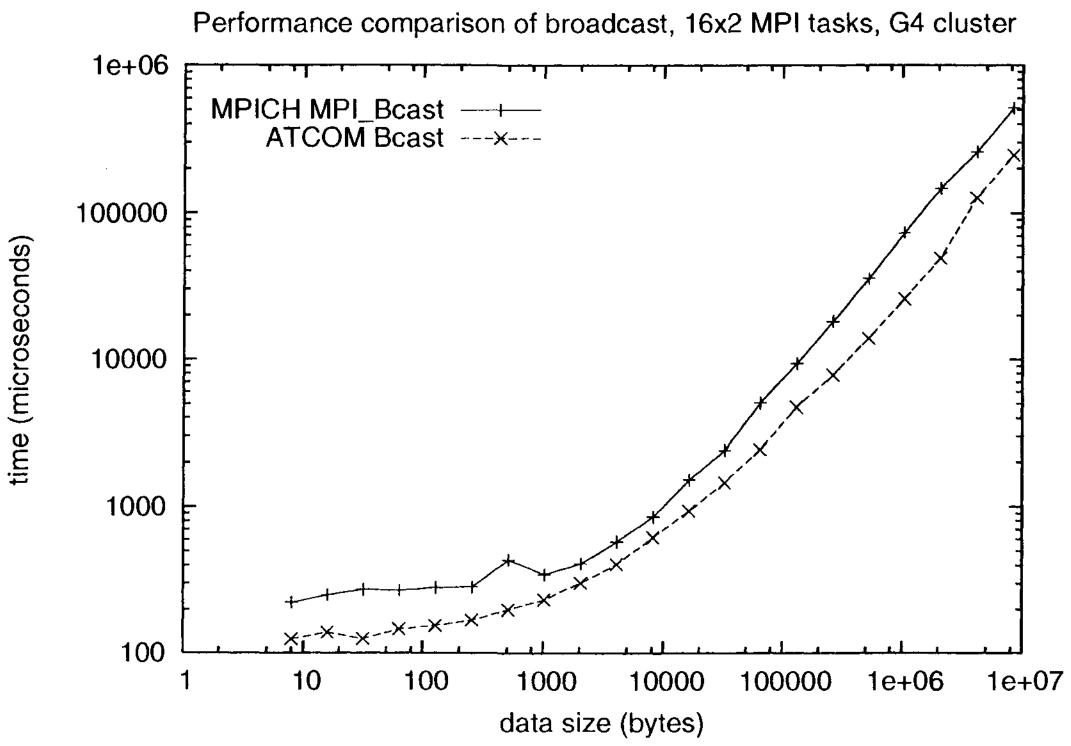


Figure 3.10 Performance comparison of broadcast on the G4 Cluster

Before we optimize a collective communication for SMP clusters we use the existing performance data to evaluate if we can expect improved performance. For example, with the all-gather operation in this section we measured the costs of shared memory gather, shared memory broadcast and intra-node broadcast algorithms. By comparing the performance results of these operations and analyzing the potential performance enhancement we could decide if all-gather should be optimized.

The algorithms for all-gather implemented in MPICH are the binomial tree algorithm for small messages and the ring algorithm for large messages. Tuning is required to determine when to use the ring algorithm and when to use the binomial algorithm. However, through experiments we found that the binomial tree algorithm is used on all three clusters for all message sizes, which leads to very bad performance especially when the message size is large. For example, on the IBM SP, an all-gather of an 8MB message on 16x16 MPI tasks using the

Message size	*IBM SP*	*Intel Cluster*	*G4 Cluster*
8	**Binomial(b, novp, shm)*	**Binomial(b, novp, shm)*	*Binomial(b, novp, msg)*
16	**Binomial(b, novp, shm)*	**Binomial(b, novp, shm)*	*Binomial(b, novp, msg)*
32	**Binomial(b, novp, shm)*	**Binomial(b, novp, shm)*	*Binomial(b, novp, msg)*
64	**Binomial(b, novp, shm)*	**Binomial(b, novp, shm)*	*Binomial(b, novp, msg)*
128	**Binomial(b, novp, shm)*	**Binomial(b, novp, shm)*	*Binomial(b, novp, msg)*
256	**Binomial(b, novp, shm)*	**Binomial(b, novp, shm)*	*Binomial(b, novp, msg)*
512	**Binomial(b, novp, shm)*	**Binomial(b, novp, shm)*	*Binomial(b, novp, msg)*
1024	**Binomial(b, novp, shm)*	*Binomial(nb, novp, msg)*	*Binomial(b, novp, msg)*
2048	**Binomial(b, novp, shm)*	*Binomial(nb, novp, msg)*	*Binomial(b, novp, msg)*
4096	**Binomial(b, novp, shm)*	*Binomial(nb, novp, msg)*	*Binomial(b, novp, msg)*
8192	**Binomial(b, novp, shm)*	*Binomial(nb, novp, msg)*	**Binary(nb, ovp, shm)*
16384	**Binary(nb, novp, shm)*	**Binary(nb, ovp, shm)*	**Binary(nb, ovp, shm)*
32768	**Binary(nb, novp, shm)*	**Binary(nb, ovp, shm)*	**Binary(nb, ovp, shm)*
65536	**Binomial(nb, novp, shm)*	**Binary(nb, ovp, shm)*	**Binary(nb, ovp, shm)*
131072	**Binomial(nb, novp, shm)*	**Binary(nb, ovp, shm)*	**Chain(nb, ovp, shm)*
262144	**Binomial(nb, novp, shm)*	**Binary(nb, ovp, shm)*	**Chain(nb, ovp, shm)*
524288	**Binary(nb, novp, shm)*	**Binary(nb, ovp, shm)*	**Chain(nb, ovp, shm)*
1048576	**Binary(nb, novp, shm)*	**Binary(nb, ovp, shm)*	**Chain(nb, ovp, shm)*
2097152	**Binary(nb, novp, shm)*	**Binary(nb, ovp, shm)*	**Chain(nb, ovp, shm)*
4194304	**Binary(nb, ovp, shm)*	**Binary(nb, ovp, shm)*	**Chain(nb, ovp, shm)*
8388608	**Binary(nb, ovp, shm)*	**Binary(nb, ovp, shm)*	**Chain(nb, ovp, shm)*

Table 3.1 Best implementations for broadcasting different message size on three platforms. The first parameter represents inter-node implementation: nb for non-blocking segmented, b for blocking. The second parameter represents if it overlaps inter-node/intra-node communications. ovp: overlap, novp:no overlap. The third parameter represents use shared memory or message passing for intra-node communication. shm: use shared memory, msg:use message passing. A * means the implementation is provided by new generic approaches.

default MPI_Allgather implementation is four times slower than MPICH's ring implementation. For this reason the performance comparison in this section is intended to show how much we can improve over the MPICH ring algorithm for all-gather operations on large messages. For small messages we also use the binomial tree algorithm.

MPICH uses MPI_Sendrecv to implement the ring all-gather; the implementation does not allow overlapping so in implementation 2 we replaced the MPI_Sendrecv with an MPI_Isend/MPI_Irecv of the whole message. Implementation 3 of all-gather extends this to the segmented non-blocking approach.

The mechanism to overlap inter-node/intra-node communication are shown in Figure 3.11.

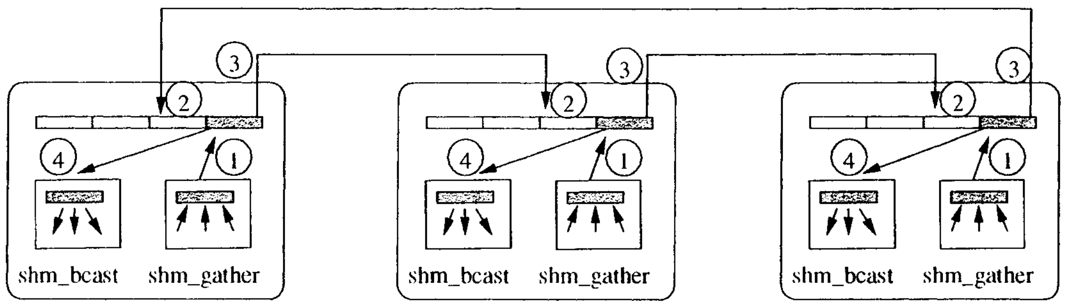


Figure 3.11 Overlapping inter-node/intra-node communications for all-gather

Each node starts with (1) a shared memory gather to the group communicator, (2) posts a non-blocking receive, (3) posts non-blocking send, and (4) starts a shared memory broadcast. It repeats (2)-(3)-(4) until the last segment is received by all processes. This is our implementation 4.

We tested all-gather of message sizes from 512K to 8M and, depending on the message size, the overall performance improvement is 18% to 69% on the IBM SP cluster, 44% to 54% on the Intel cluster, and 11% to 69% on the G4 cluster. Figure 3.12 and 3.13 show the performance improvements by adding layers of optimization on the three platforms.

From the results, we can observe the following:

1. Existing generic techniques to optimize collective communications do not provide optimal performance on SMP clusters. By taking the SMP architecture into account, the performance of collective communications can be significantly improved by new generic optimizations.

2. Using shared memory collective communications is the key to performance improvement on SMP clusters. It reduces the latency and allows overlapping of inter-node/intra-node communications.

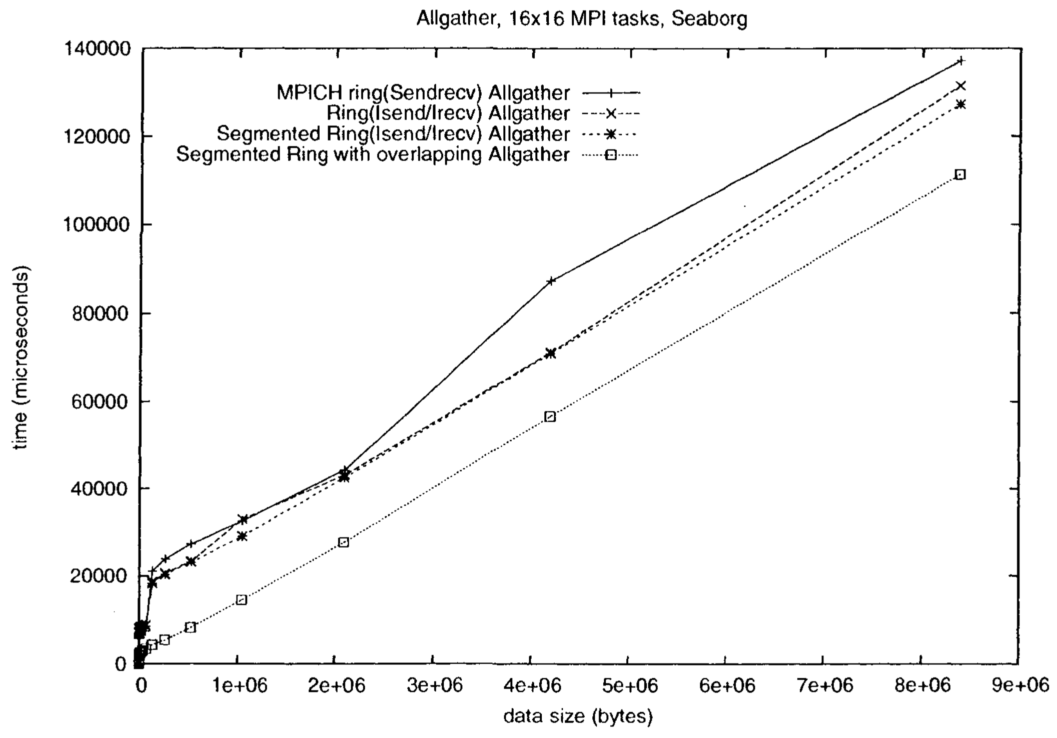
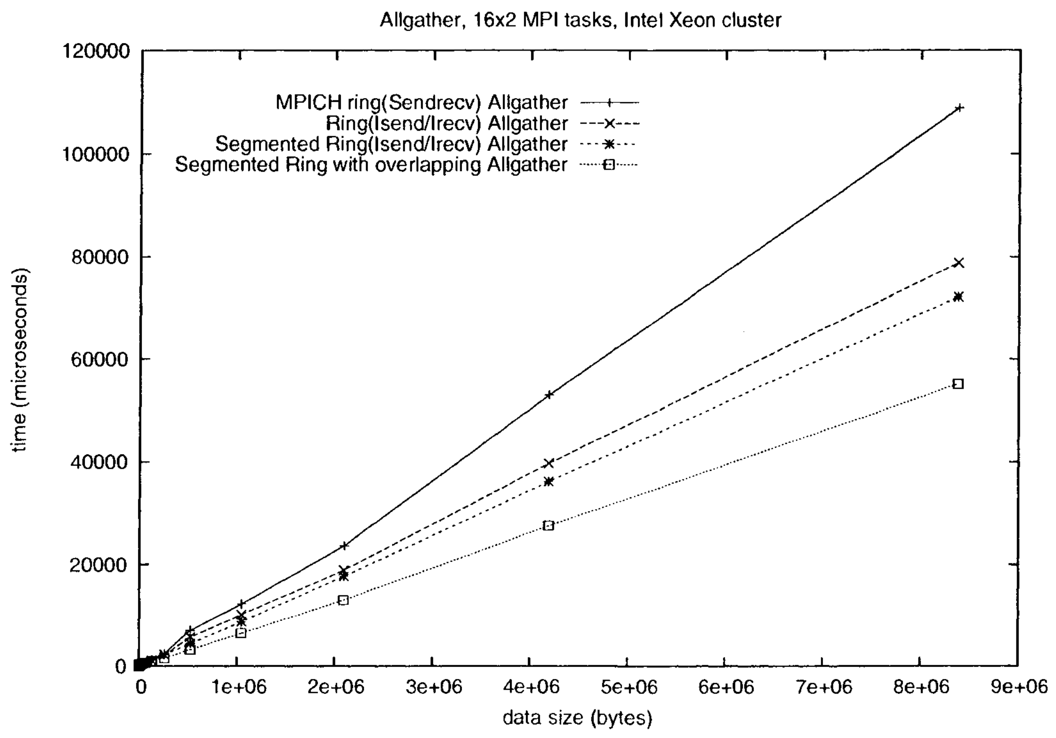


Figure 3.12 Performance comparison of all-gather on the IBM Cluster

3.5 Summary

In this chapter we have shown that it is possible to design generic implementations of collective operations that take advantage of both shared memory collective communications and overlapping inter-node/intra-node communications. Several collective communications are implemented and our experimental results show that, after proper tuning, the performance improvement over the existing implementations is significant on three different SMP clusters. While collective communication algorithms on the inter-node layer may be regarded as exhausted, by taking the SMP architecture into account and using implementation techniques generically available on all platforms, we still can see impressive performance improvement even if each SMP node has only two processors. This kind of collective communications on SMP clusters are composed with shared memory collective communications on the intra-node layer and point-to-point communications on the inter-node layer; we call them mixed mode



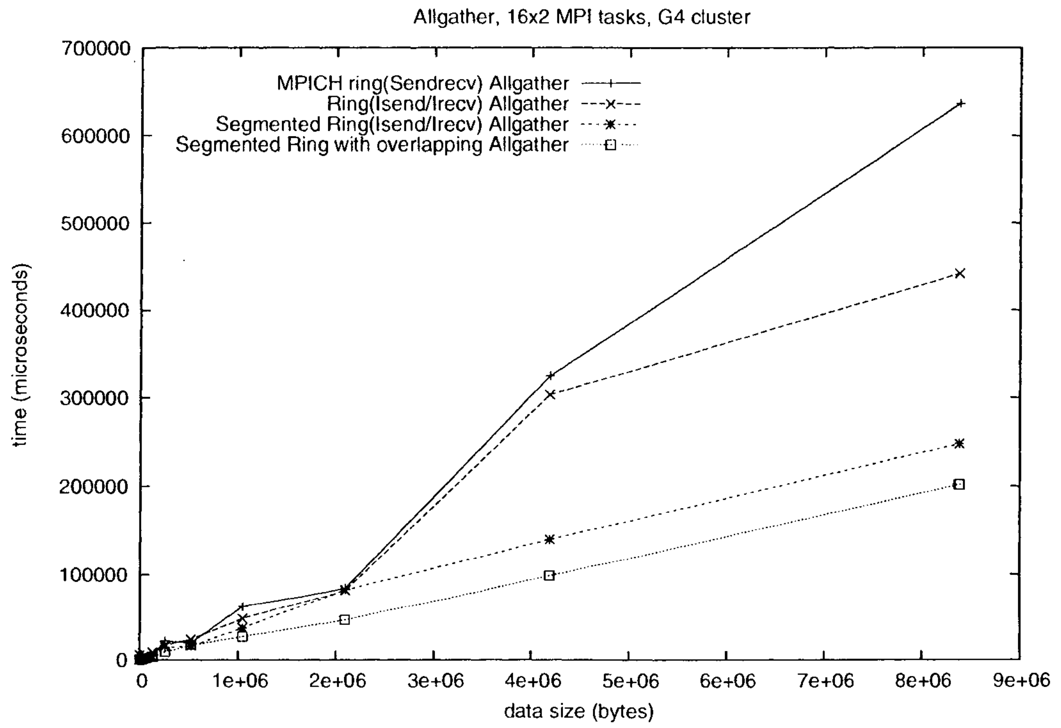


Figure 3.13 Performance comparison of all-gather on the Intel Xeon Cluster and the G4 Cluster

collective communications.

The methods developed in this chapter are based on MPI blocking send/receive and non-blocking send/receive. If a different message passing library provides the same functionalities for point-to-point communications, they can be used to replace the MPI functions we are using in the current implementations.

The price of portability to achieve good performance is that many parameters require tuning. We provide a framework to design mixed mode collective communications on SMP clusters in this chapter, and in the next chapter we will describe a new performance model that extends existing performance models to describe the characteristics and predict the performance of mixed mode collective communications. Several tuning strategies that are developed based on the new performance model will also be discussed.

CHAPTER 4. PERFORMANCE MODEL AND TUNING STRATEGIES

4.1 Introduction

In the previous chapter we have developed new approaches to utilize the architecture of SMP clusters for collective communications. However, our approaches require the tuning of many parameters. Without a good tuning mechanism, the only tuning method is to exhaustively test every setting and find the best one. Such exhaustive tuning process is very time consuming and is not realistic to be used in practical systems.

An approach to facilitate automatic tuning process, which is taken by MagPIe and ACCT, is to use a performance model to predict the performance of collective communication implementations, and based on the prediction results select a certain range of data for run time testing. The performance model of ACCT is based on parameterized LogP model used by MagPIe, which in turn is based on the other communication models. In this chapter we first introduce several existing performance models for parallel communications, then we discuss our performance model that covers overlapping of inter-node/shared memory collective communications. We show how this model can describe the characteristics of mixed mode collective communications, and how to utilize it to develop tuning mechanisms for mixed mode collective communications.

4.2 Performance Models for Parallel Communications

There are some existing parallel programming models that are fundamentally different from the parallel model we are using to design collective communications in this dissertation. For example, BSP model [2] also divides a parallel algorithm into computation and communication stages. However, in BSP model every communication stage is an *all-to-all* communication instead of point-to-point communications as in the other models. Every performance model discussed in this section is based on point-to-point communications.

4.2.1 The Hockney Model

Figure 4.1 shows the most commonly used performance model, the Hockney model [1] (sometimes it is called permutation model). This model uses four parameters: startup latency α, the inverse transmission rate β, the message size m, and the total number of processors p. Under this performance model, the cost of using flat-tree algorithm to broadcast a message of size m is *(p-1)* * (α + β * m) since the root needs to send the message *p-1* times to *p-1* processes, and each time it costs (α + β * m). A binomial tree broadcast of a message of size m thus costs *logp* * (α + β * m).

This model is easy to be understood, but it is too simplified to describe different implementations of a collective communication, and can be used only when every collective communication is implemented with blocking calls. When a collective communication is latency bound, blocking implementations usually perform better than non-blocking implementations and the Hockney model can provide very good predictions for their performance. However, when a collective communication is bandwidth bound, pipelining may be required to achieve optimal performance and non-blocking calls are usually used to implement the pipelined version of a collective communication algorithm. Under this case the Hockney model is not suitable for performance prediction.

4.2.2 The LogP Model

The LogP model [3], as shown in Figure 4.2, introduces one more parameter g, the gate value, and the meanings of some parameters are slightly different from the Hockney model. In the LogP model, there are four parameters: L, the latency for sending a message between two processes; o: the overhead for sending or receiving a message; g: the time that the network is occupied, i.e., a sender can not send a message within g time after the previous message is sent, and P, the number of processes. Figure 4.2 shows the cost of sending a message from one

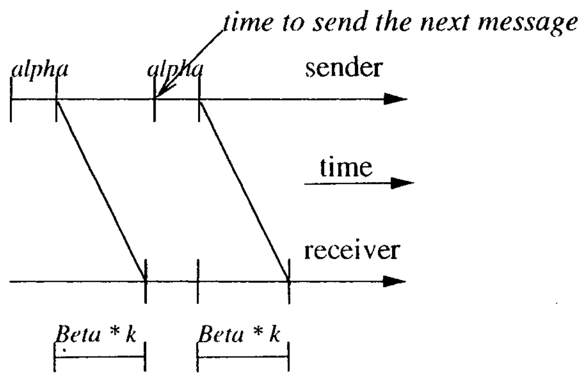


Figure 4.1 The Hockney model. Sending a message of k bytes costs α + $\beta * k$

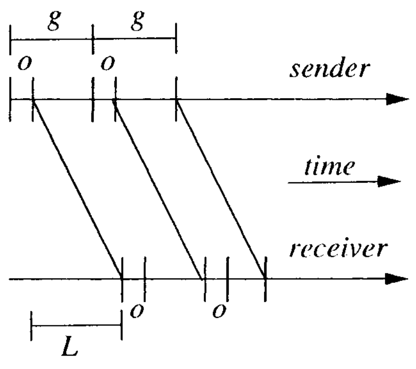


Figure 4.2 The LogP model. Sending a message of k bytes costs o + $(k\text{-}1)*max\{o,g\}$ + L + o

process to another process under LogP model.

Originally the LogP model was proposed as a model that allows overlapping of communication and computation (the time for overlapping is g - o); however, due to its complexity for designing parallel applications, LogP model is often used as a performance model for designing communication algorithms instead of designing parallel applications.

4.2.3 The LogGP Model

The LogP model assumes a message is sent in the minimum data type of a particular machine, making it good for measuring communications of small message size. When sending a large message, the prediction of the LogP model is not accurate since a message can be sent

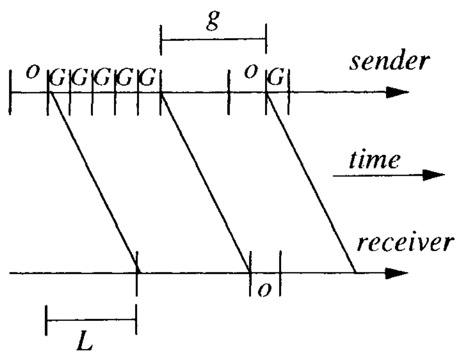


Figure 4.3 The LogGP model. Sending a message of k bytes costs $o + (k\text{-}1)*G + L + o$

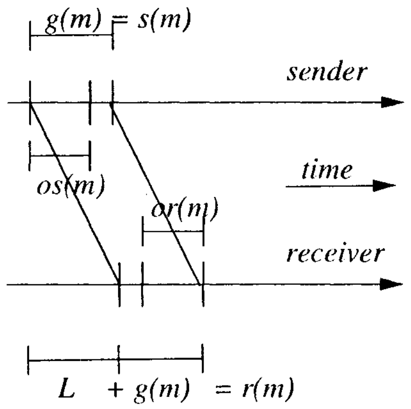


Figure 4.4 The Parameterized LogP (P-LogP) model.

in a large chunk in modern high speed network instead of just the minimum data type. To remedy this inaccuracy, the LogGP [4] model was proposed. Another parameter G is added into the LogP model that stands for gate value per byte for large messages. When the size of a message is small, the LogGP model is the same as the LogP model. Figure 4.3 shows the cost of sending k bytes in LogGP model.

4.2.4 The Parameterized LogP Model (P-LogP Model)

The LogGP model assumes a linear model for calculating G. The work by Kielmann [10] shows that this is not the case in real applications since the value of G changes with the communicating message size. Thus the parameterized LogP model introduces $g(m)$, which stands for the time a sender has to wait to send the next message of size m. The P-LogP model also distinguishes the sender overhead $os(m)$ from the receiver overhead $or(m)$, while in LogP or LogGP model o stands for the overhead of both sender and receiver. The meaning of L also is slightly different than as in LogP or LogGP since in the P-LogP model L includes the overhead os. Both LogP and LogGP models assume there is only one layer of communication, thus is not suitable for communications with hierarchy. The Parameterized LogP model was designed targeting for cluster of WAN, which have communication hierarchy and the latency and bandwidth between two layers of communication are at least two order of magnitude. By measuring the cost of $os(m)$, $or(m)$, *g(m) and* L on different layers, it can deduce the overlapping degree for a collective communications. When applying this model into cluster of SMPs, it can be used only when the intra-node communications are done with message passing. Since we are using shared memory collective communications on the intra-node layer, we can not use parameterized LogP model to predict the performance of our collective communication implementations. The parameterized LogP model is shown in figure 4.4.

4.2.5 The Other Performance Models

At least two performance models attempt to model network contention, the C^3 model [5] and the LoGPC model [7]. Both models made assumptions on the restrictions of the pattern of network contention. MPI assumes fully connected network between compute nodes, and we also use this assumption to design our performance model.

The models introduced in this section are by no mean complete. There are many different communication models for different purpose, and we introduce those that are most related to our research.

4.3 The Simplified Programming Model

When designing our collective communications, we mixed shared memory collective communications with point-to-point communications. Since there are many performance models for point-to-point based collective communications, and our mixed mode collective communica-

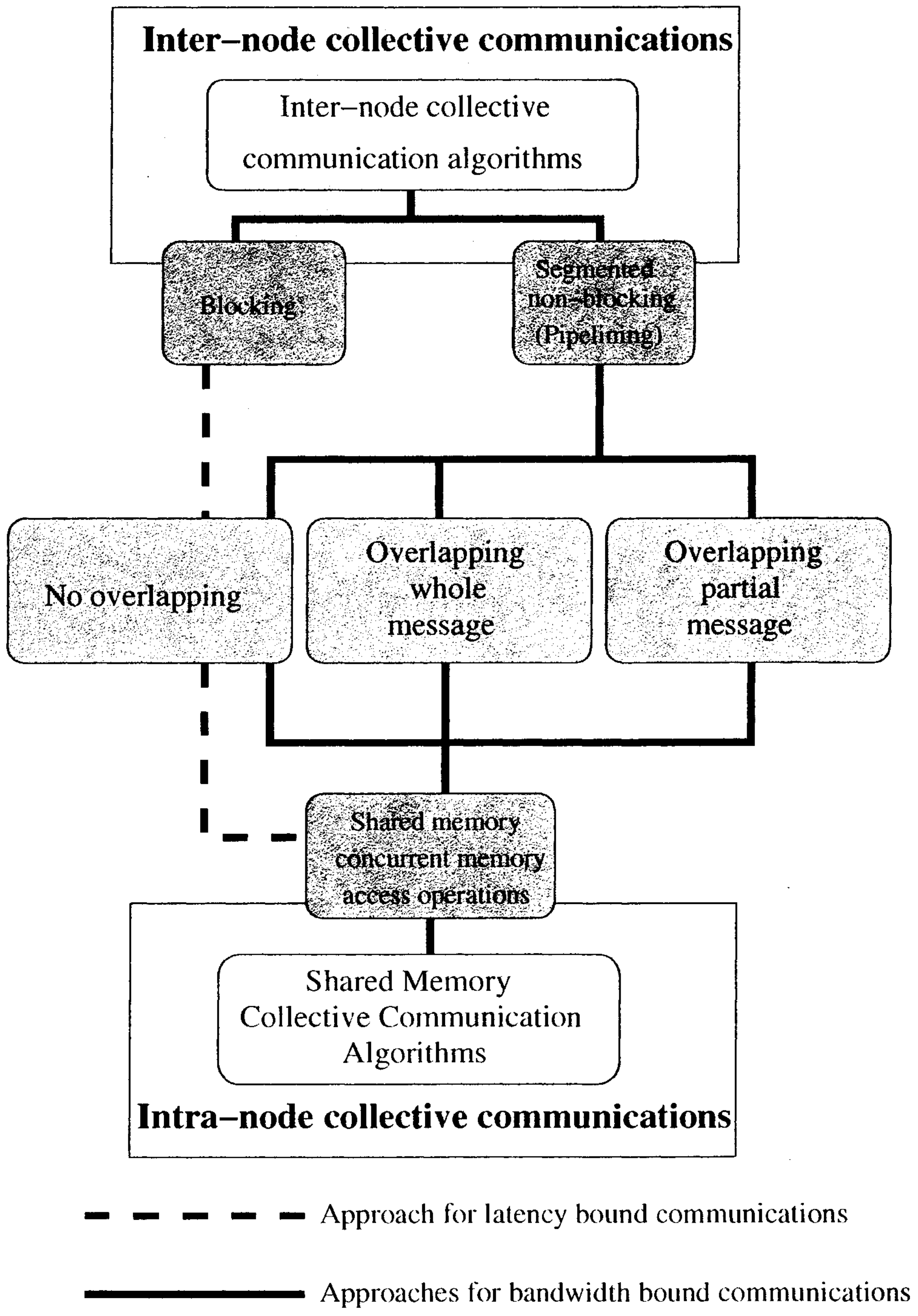

Inter–node collective communications
Inter–node collective communication algorithms
Blocking
Segmented non-blocking (Pipelining)
No overlapping
Overlapping whole message
Overlapping partial message
Shared memory concurrent memory access operations
Shared Memory Collective Communication Algorithms
Intra–node collective communications
Approach for latency bound communications
Approaches for bandwidth bound communications

Figure 4.5 The simplified programming model of mixed mode collective communications on SMP clusters.

tions start with inter-node, point-to-point based collective communications followed by shared memory collective communications, our approach is to extend one of the existing performance models to cover mixed mode collective communications.

The programming model described in Figure 3.2 lists every possible approach to design a collective communication on an SMP cluster. It is not practical to design a collective communication that follows every path in Figure 3.2. Moreover, a different path may need a different performance model to predict the communication latency. In this chapter we use a simplified programming model that shows possible paths for latency bound communications and bandwidth bound communications, and the performance models we will be discussing are for the paths in this simplified model. Figure 4.5 outlines the simplified programming model of mixed mode collective communications on SMP clusters.

As described in the previous chapter, there are three layers: the inter-node communication layer, the intra-node communication layer, and the overlapping mechanisms. Each SMP node has one process (group coordinator) in charge of communications with the other nodes. A collective communication is processed in two stages: first communications are processed between the group coordinators on the inter-node layer, then on the intra-node layer the communications are processed between a group coordinators and the processes within the same node. For collective operation such as gather, the order of communications is reversed.

4.3.1 Latency Bound Communications

Blocking send/receive incur less software overhead than non-blocking send/receive; the major advantage of shared memory collective communications is low latency for small message, as shown in one of our work [113]. Thus for latency bound communications [39], we use blocking implementations on the inter-node layer, combined with shared memory collective communications on the intra-node layer without overlapping. This is the dashed line in Figure 4.5.

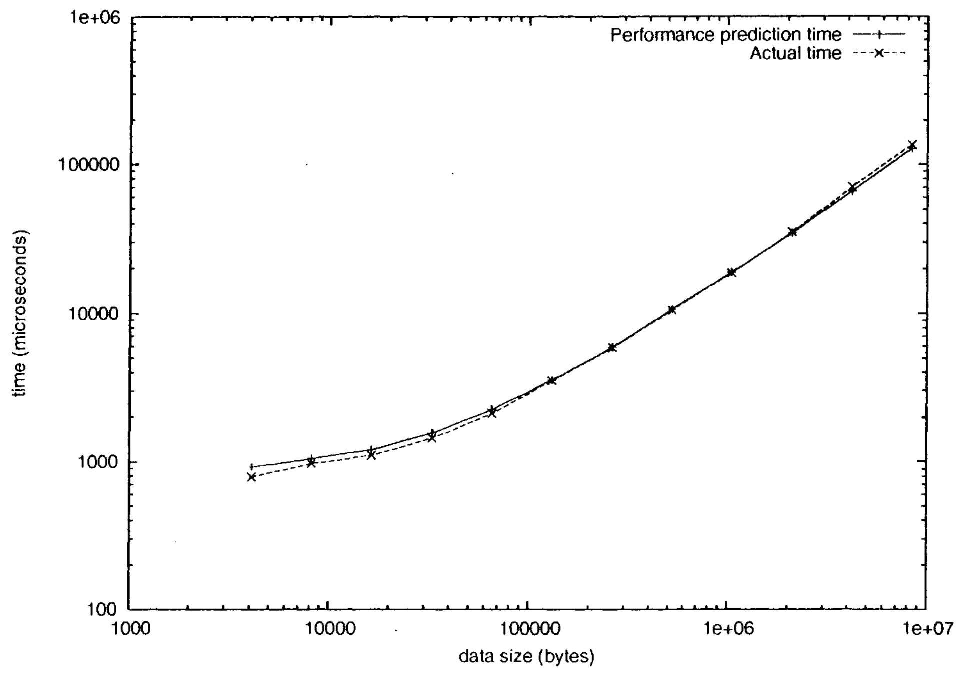


Figure 4.6 The performance prediction and actual time of mixed mode binary tree broadcast without overlapping, using 8x16 MPI tasks on the IBM cluster.

4.3.2 Bandwidth Bound Communications

For bandwidth bound communications, we use segmented non-blocking implementations on the inter-node layer to take advantage of pipelining. Using non-blocking calls also make it possible to design mechanisms to overlap inter-node/intra-node communications.

The overlapping mechanism is the same as described in the previous chapter. There are three possible overlapping mechanisms: overlapping a whole message segment (total overlapping), overlapping partial message segment (partial overlapping) and no overlapping at all. They are the solid lines in the center of Figure 4.5. We have discussed the reason to take total overlapping and partial overlapping approaches in the previous chapter. There are times when no overlapping may provide the best performance and we also include it in the simplified programming model.

4.3.3 Performance Modeling of Non-overlapped Approaches

When there is no overlapping between inter-node/intra-node communications, the performance prediction is straightforward: the cost of the inter-node communications plus the cost of the shared memory collective communications. We can measure the cost of collective communications on each layer individually then add them together.

Figure 4.6 shows the comparison of this prediction and the actual run time of binary tree broadcast. We also measure the performance of the other algorithms and the results show that the prediction of this approach is very close to actual run times. Since there is no overlapping, we only need to tune the inter-node layer implementations (segmented non-blocking) to achieve the optimal performance. In this case we can use the methods by Kielmann *et al.* [11] or Vadhiyar *et al.* [6] to reduce tuning time.

4.3.4 Performance Modeling Issues of Overlapping Approaches

For the rest of this chapter, "mixed mode collective communications" refers to mixed mode collective communications **with overlapping** unless specified otherwise. "Inter-node collective communications" means the **pipelined** version of collective communications on the inter-node layer, between group coordinators only, and without any shared memory collective communication.

For mixed mode collective communications, since there is no performance model to describe this type of collective communications, our only choice was to exhaustively examine every possible setting. With total overlapping, the number of tests required to tune a mixed mode collective communication is the same as tuning an inter-node collective communication. For example, to tune an inter-node chain-tree broadcast of 8MB message, if the testing starts with segment size = 4KB, and double it each step, we need to examine 4KB, 8KB, ..., 4MB, 8MB, a total of 12 tests. For mixed mode chain tree broadcast with whole segment overlapping, we also need 12 tests and the testing time is much longer.

When we allow partial overlapping, the number of tests required for tuning is much larger. For a collective communication we need to find the best {*segment size, overlapping size*} pair

	16K	32K	64K	128K	256K	512k	1M	2M
1024	0.19	0.20	0.17	0.16	0.16	0.15	0.15	0.13
2048	0.18	0.19	0.17	0.16	0.16	0.15	0.15	0.17
4096	0.18	0.18	0.17	0.16	0.16	0.15	0.15	0.17
8192	0.18	0.18	0.16	0.15	0.16	0.15	0.15	0.17
16384	0.16	0.18	0.16	0.15	0.16	0.15	0.15	0.18
32768	-	0.16	0.15	0.14	0.15	0.15	0.16	0.18
65536	-	-	0.14	0.14	0.15	0.15	0.16	0.18
131072	-	-	-	0.14	0.16	0.16	0.16	0.19
262144	-	-	-	-	0.16	0.17	0.18	0.20
524288	-	-	-	-	-	0.18	0.21	0.22
1048576	-	-	-	-	-	-	0.23	0.27
2097152	-	-	-	-	-	-	-	0.35

Table 4.1 A performance matrix of mixed mode collective communications.

for every message size to achieve the optimal performance. If we list all possible combinations of this pair, using power of two for both segment size and overlapping size, we can form a performance matrix as shown in Table 4.1. The columns represent segment size, and the rows represent overlapping size. In Figure 4.1 the segment size for testing are from 16KB to 2MB, and the overlapping size are from 1KB to 2MB. An entry in the performance matrix can be any data of interest. For example, in Table 4.1, entry (2,5) in the performance matrix shows the performance for broadcasting a 2MB message, using a segment size of 256KB on the inter-node layer, and the overlapping size for a segment is 2KB. Since the overlapping size is always less than the segment size, only the entries on the upper-right section of the performance matrix have values.

For a mixed mode collective communication, every message size has one such performance matrix indicating the number of tests needed for tuning. Without a proper performance model, we are forced to examine every possible entry in the performance matrix. If there are m rows and n columns in a performance matrix, we need to examine at least $(m*n)/2$ entries just for a certain message size, under a particular setting. Clearly, the number of tests required is large and the tuning time is not acceptable. Without a proper tuning mechanism as part of the support infrastructure it is not practical to use the programming model in Figure 4.5.

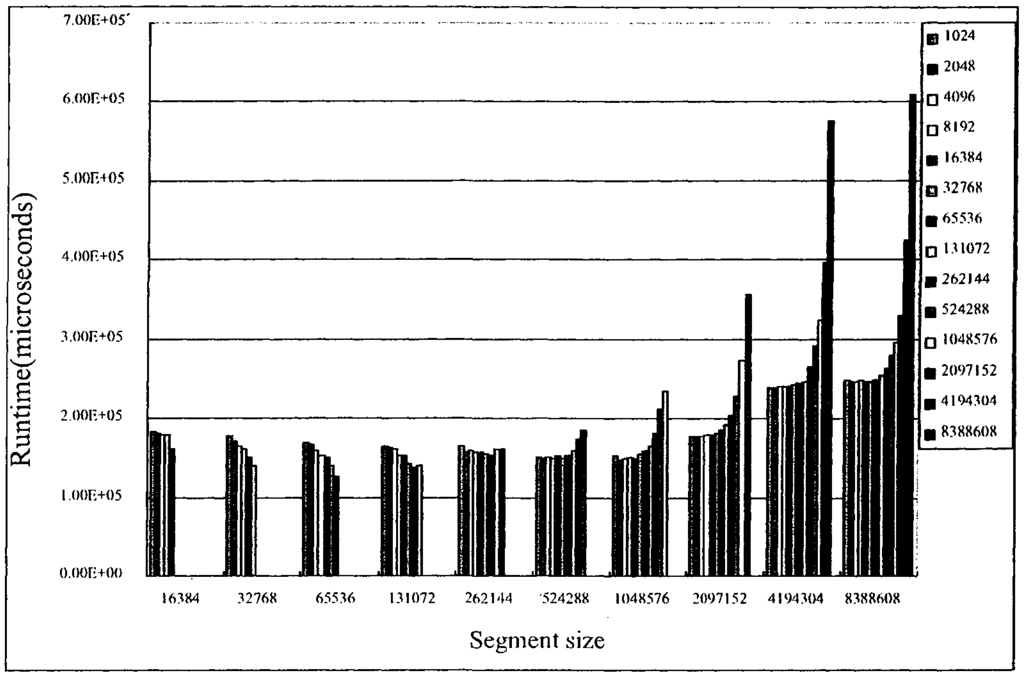


Figure 4.7 The performance of mixed mode chain tree broadcast of 8MB message on the IBM cluster (8x16 MPI tasks).

4.4 The Characteristics of Mixed Mode Collective Communications

To model the performance of mixed mode collective communications, we first address the following question: when tuning a mixed mode collective communications, given different {*segment size, overlapping size*} pairs, if we fix segment size and vary overlapping size, how does the performance change with the increase of overlapping size.

4.4.1 Experimental Results of Mixed Mode Chain Tree Broadcast

Figure 4.7 and 4.8 show the performance of broadcasting 8MB messages using mixed mode chain-tree on the IBM cluster (8x16 MPI tasks) and on the Intel cluster (8x2 MPI tasks). The tests that use the same segment size are put together in a group, and different bars in a group represent different overlapping sizes. The the height of a bar representing the run time of that setting. For example, for the group of 1MB, it means the segment size is 1MB for broadcasting 8MB message. The bars within 1MB group from left to right represent overlapping size of a 1MB message is 1KB, 2KB, and so on up to 1MB.

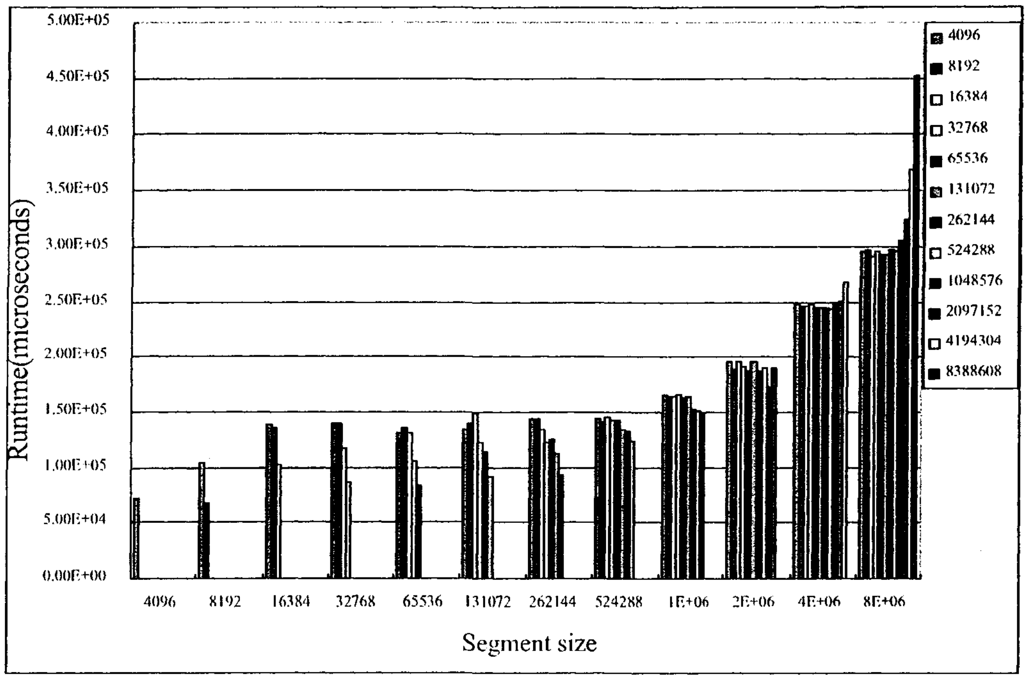


Figure 4.8 The performance of mixed mode chain tree broadcast of 8MB message on the Intel Xeon cluster (8x2 MPI tasks).

From Figures 4.7 and 4.8, we divide the performance curves into three different types: The first type is when performance improves roughly with the increase of overlapping size. On the IBM cluster this is up to 128KB of segment size; on the Intel cluster it is up to 1MB. The second type is when the performance is irregular, neither always increasing nor always decreasing with the increase of overlapping size. The segment size of 256KB on the IBM cluster and 2MB, 4MB on the Intel cluster belong to this type. The third type is when the performance decreases with the increase of overlapping size; this happens on the IBM cluster when the segment size is larger than 256KB, or on the Intel cluster when the segment size is larger than 4MB. Apparently, for mixed mode collective communications, the performance does not always get better or get worse with the increase of overlapping size.

The best segment size for inter-node collective communications does not always provide the best performance for mixed mode collective communications. In Figure 4.9, the best segment size for the inter-node chain tree is 1MB, but for mixed mode chain tree the best segment size is 64KB. Also, the performance of mixed mode collective communications (more processes) may

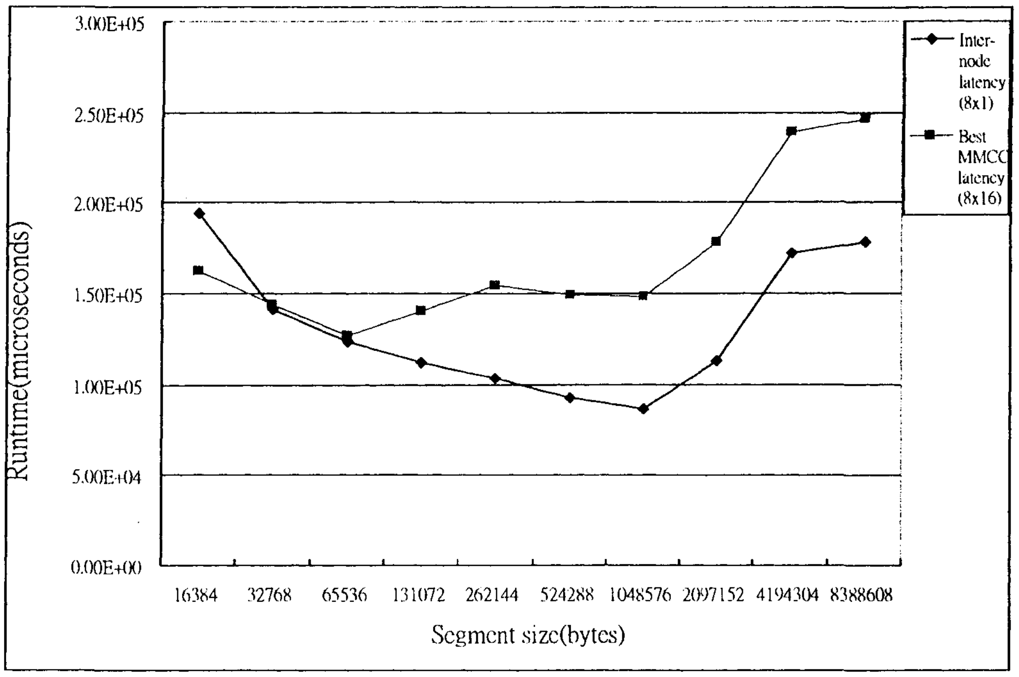


Figure 4.9 The performance curves of mixed mode chain tree broadcast (8x16 MPI tasks) and inter-node chain tree broadcast (8x1 MPI tasks) of 8MB message on the IBM cluster.

be better than inter-node collective communications (fewer processes), such as with a segment size of 16KB in Figure 4.9.

4.5 Performance Modeling

The major challenge in designing a performance model for mixed mode collective communications is to include memory operations into the model. In previous chapters, a shared memory collective communication is composed of several shared memory operations. Each shared memory operation can in turn be decomposed to memory operations as detailed as operations involving cache coherence mechanisms. However, a performance model for collective communications that goes into such details is too complicated and is not preferable.

We designed our shared memory collective communications based on a generic shared memory architecture [113, 115]. Our performance model thus is also on top of this generic SMP architecture. We look at the total cost of a shared memory collective communication

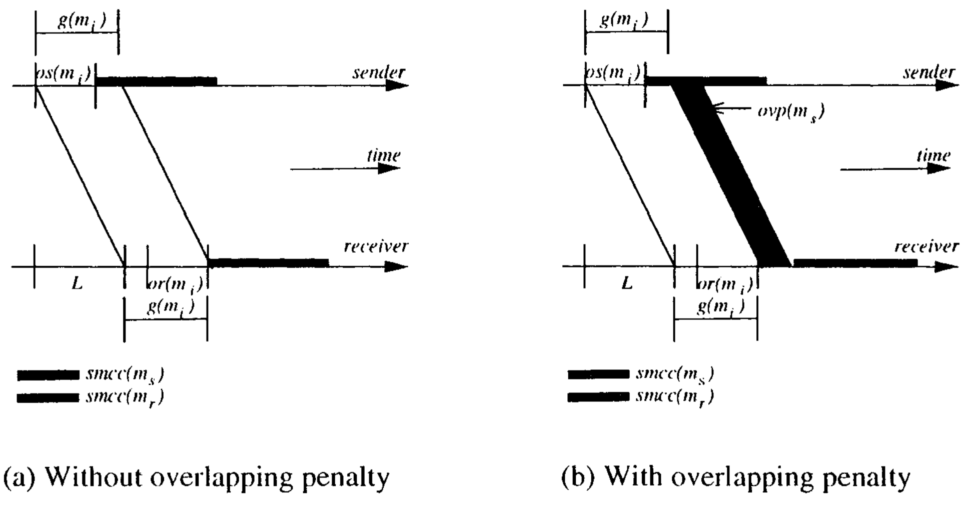


Figure 4.10 The performance model for mixed mode collective communications.

instead of the composition of shared memory operations, thus the cost of a shared memory collective communications is primarily a function of message size.

Among the existing performance models, the parameterized LogP model [11] has many parameters that are also functions of message size. It also provides the tools to measure those parameters. Since the cost of a shared memory collective communication is a function of message size, we decided to develop our model base on the parameterized LogP model.

4.5.1 The Performance Model

The parameters we use in this model are as follows.

p: number of nodes.

L: latency of sending a message from one process to another process on the inter-node layer.

M: total message size.

m_i: the size of a message segment for the inter-node collective communications.

k: total number of segments, it equals M/m_i.

$g(m_i)$: gate value, also means the minimum interval between sending or receiving two consecutive messages of size m_i on the same node.

$os(m_i)$: software overhead of sending a message of size m_i.

$or(m_i)$: software overhead of receiving a message of size m_i.

m_s: data size of the shared memory operation that is overlapping with inter-node communications.

m_r: data size for the remaining message segment to be processed with the last shared memory operation. $m_r = M - m_s * k$.

$smcc(m_s)$: the cost of shared memory operation of message of size m_s.

$smcc(m_r)$: the cost of the last shared memory operation.

$ovp(m_s)$: overlapping penalty for overlapping inter-node/shared memory collective communications, when the overlapping size is m_s.

$f(M, m_i, m_s)$: total latency of the operation of size M, with inter-node layer segment $= m_i$, and overlapping size for each segment $= m_s$.

Figure 4.10(a) outlines sending a segment from one node to another node in this model. The sender spends $os(m_i)$ overhead on sending a message segment of size m_i. It takes $g(m_i)$ time to send the segment of size m_i. After $os(m_i)$ it starts the intra-node collective operations of message size m_s, and this takes $smcc(m_s)$ time. If m_s is less than m_i, it will require another $smcc(m_r)$ on the remaining segment to complete the operation. Thus the total cost on the sender is $max(g(m_i), os(m_i) + smcc(m_s)) + smcc(m_r)$. The receiver waits L to receive the first byte of the message and $g(m_i)$ to receive the whole message segment, then it starts shared memory collective communication.

What makes mixed mode collective communications less predictable is the cost due to overlapping of the two communication layers, the overlapping penalty $ovp(m_s)$. It is possible that the segment size and the overlapping size are both very large, and the bandwidth of memory bus can not sustain this high amount of data traffic. For example, if the sender posts a non-blocking send of a 8M message then proceeds with shared memory communications of 8MB message, it is possible that the MPI layer can not even access the data in the memory because the memory bus is occupied by shared memory collective communications; the MPI

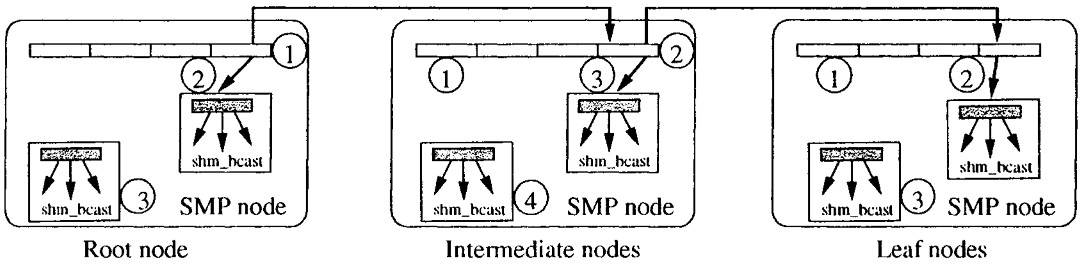


Figure 4.11 The overlapping mechanism of chain-tree broadcast.

layer has to wait until it has access to the memory bus. Thus with overlapping penalty the receiver can not receive the whole message segment until *L + g(8MB) + ovp(8MB)*. Through the experiments we found that when the overlapping size is m_s for a message segment, the overlapping penalty can be as high as $smcc(m_s)$. The receiving node can not start shared memory operation until the whole segment is received. The cost in Figure 4.10(a), when there is no overlapping penalty, is:

$$f(m_i, m_i, m_s) = L + g(m_i) + smcc(m_s) + smcc(m_r) \tag{4.1}$$

and in Figure 4.10(b), when there is overlapping penalty, is:

$$f(m_i, m_i, m_s) = L + g(m_i) + ovp(m_s) + smcc(m_s) + smcc(m_r) \tag{4.2}$$

4.5.2 Example: Chain Tree Broadcast

For ease of presentation, we only discuss the chain tree broadcast as an example in this chapter. The other algorithms and collective operations can be derived via a similar approach. Also, the performance equations in this section focus on how much "extra cost" is added on the inter-node collective communications when we allow overlapping. The cost equations for inter-node collective communications are discussed in other work [11, 6] and we do not repeat that analysis here.

Figure 4.11 shows the mechanism of mixed mode chain tree broadcast. It is the same as the one in Figure 3.4. We repeat it here for easy of reading and comparison.

Figure 4.12 shows applying the performance model on mixed mode chain-tree broadcast of three nodes, corresponding to a root node, an inter-mediate node and a leaf node.

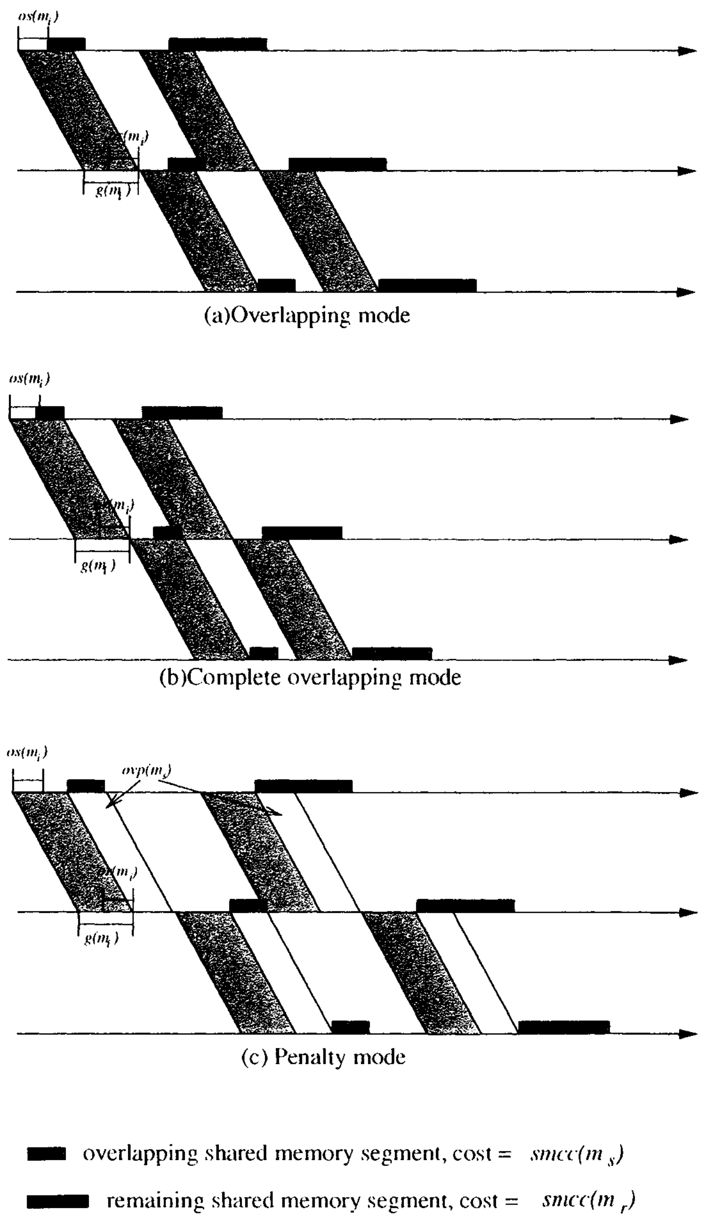

os(m_i)
g(m_i)
(a)Overlapping mode
os(m_i)
g(m_i)
(b)Complete overlapping mode
os(m_i)
ovp(m_i)
g(m_i)
(c) Penalty mode
overlapping shared memory segment, cost = smcc(m_s)
remaining shared memory segment, cost = smcc(m_r)

Figure 4.12 The performance model of mixed mode chain tree broadcast under different modes. (a) Overlapping mode, a shared memory collective communication may incur extra cost for sending/receiving messages. (b) Complete overlapping mode, the theoretical lower bound, when the cost of every shared memory collective communication can be hidden by overlapping. (c) Penalty mode, the theoretical upper-bound, when a shared memory collective communication cause the maximum amount of overlapping penalty($smcc(m_s)$).

In Figure 4.12(a), the cost of a shared memory collective communication can not be totally hidden by the overlapping mechanisms. It incurs extra cost because *smcc(m_s)* is greater than *($g(m_i)$ - $os(m_i)$)*, and will cause longer interval time between sending two message segments. This interval can be increased from $g(m_i)$ to *os(m) +* $smcc(m_s)$. Since there are *k-1* intervals, the total cost is:

$$\begin{aligned} f_1(M, m_i, m_s) &= \mathcal{L} + ((os(m_i) + smcc(m_s)) - g(m_i)) \\ &\quad *(k-1) + smcc(m_s) + smcc(m_r) \end{aligned} \tag{4.3}$$

$\mathcal{L}$ represents the cost of inter-node chain tree broadcast. The term *smcc(m_s)* is the cost of shared memory operation for the last "overlapping" segment on the last node since the cost of this shared memory broadcast can not be hidden. The number of segments k cannot be too small so the cost of the last *smcc(m_s)* can be amortized through pipelining and overlapping.

Figure 4.12(b) shows the theoretical lower bound of mixed mode chain tree broadcast. In this case the cost of every shared memory operation can be hidden by overlapping inter-node/intra-node communications (complete overlapping mode), thus the total cost is :

$$f_2(M, m_i, m_s) = \mathcal{L} + smcc(m_s) + smcc(m_r) \tag{4.4}$$

Figure 4.12(c) shows the theoretical upper-bound of mixed mode chain tree broadcast. In this case overlapping inter-node/shared memory collective communications causes overlapping penalty (penalty mode). There are three places where run time are increased:

1. The cost of sending a message segment of size m_i can be increased from *g(m_i)* to *g(m_i) + ovp(m_s)*. There are k segments, thus there is *k * ovp(m_s)* extra cost.

2. The interval between sending two segments is extended to *g(m_i) + ovp(m_s)*. There are *k-1* intervals, so another extra cost of *(k-1) * ovp(m_s)*.

3. The time for the first byte to arrive the last node is an additional *ovp(m_s) * (p-2)*.

The total cost under penalty mode is:

$$\begin{aligned} f_3(M, m_i, m_s) &= \mathcal{L} + ovp(m_s) * k \\ &\quad + ovp(m_s) * (k-1) + ovp(m_s) * (p-2) \\ &\quad + smcc(m_s) + smcc(m_r) \end{aligned} \tag{4.5}$$

If we use *smcc(m_s)* as the maximum overlapping penalty, equation 4.5 becomes:

$$\begin{aligned} f_3(M, m_i, m_s) &= \mathcal{L} + smcc(m_s) * k \\ &\quad + smcc(m_s) * (k-1) + smcc(m_s) * (p-2) \\ &\quad + smcc(m_s) + smcc(m_r) \end{aligned} \tag{4.6}$$

4.5.3 Prediction Results and Discussion

The costs of shared memory broadcast *smcc(m)* and gate values *g(m)* on the IBM cluster are shown in figure 4.13. Gate values and send overheads are acquired using the tool provided by MagPIe [12]. We plug in *smcc(m_s), smcc(m_r), g(m_i)* and *os(m_i)* into equations 4.3, 4.4 and 4.6 to calculate broadcast 8MB message on the IBM cluster(8x16 MPI tasks), the show the results in Figure 4.15, 4.16, and 4.17.

Table 4.2 is the performance matrix with the number in an entry indicates which equation provides the best prediction for that entry. A * indicates that all three equations give close predictions with error less than 5%. Clearly, when the ratio of *(segment size)/(overlapping size)* is large enough, *smcc(m_i)* is insignificant compared with *smcc(m_r)*. In these cases the inter-node latency plus *smcc(m_r)* dominates the performance and the predictions of three equations are very close.

Table 4.3 shows the error of the prediction equation in Table 4.2. A negative number indicates "under estimate" while a positive number means "over estimate".

On the IBM cluster using 16 processors per node, the cost of shared memory broadcast *smcc(m)* is much larger than the gate value *g(m)* for most message size as shown in Figure

4.13. We speculated that equation 4.3 should give better predictions when it is not under penalty mode. However, equation 4.4 (complete overlapping mode) provides the best predictions up to segment size equals 128K. Within this range even complete overlapping equation (the theoretical lower bound) over estimates the performance and the error of predictions are are all higher than 10%. We will discuss the reason in the next subsection.

Equation 4.3 is the best prediction only for segment size 256KB and 512KB. When the segment size is larger than 512KB, penalty mode equation 4.6 gives predictions very close to actually run time. The equation also predicts that within this range, the performance is decreased with the increase of overlapping size, as shown in Figure 4.17.

Given a {*segment size, overlapping size*} pair, if it is under complete overlapping mode, we want the overlapping size as large as possible. The performance thus can increase with the increase of overlapping size. The experimental results and our performance equation both reflect this characteristics. Once it is not in complete overlapping mode, the performance may vary according to *smcc(m)* and *g(m)*. If it is in penalty mode, the performance decrease with the increase of overlapping size.

It is possible that a segment may be in overlapping mode when the overlapping size is small. When the overlapping size is increased up to a point, it switches to penalty mode. In that case we can see a performance increase curve followed by a decreasing curve, which is also similar to the second type of curve we found in some experimental results.

On the Intel cluster, $g(m_i)$ - $os(m_i)$ is about two times of $smcc(m_i)$ as shown in figure 4.14, and equation 4.4 gives the best predictions most of time except when the segment size and the overlapping size are both large than 4M; the penalty mode equation gives better predictions in that case.

4.5.4 Limitations of the Performance Model

There are following limitations of this model:

1. It provides good predictions only when the latency of mixed mode communications is higher than the inter-node collective communications.

	16K	*32K*	*64K*	*128K*	*256K*	*512k*	*1M*	*2M*	*4M*	*8M*
1024	*4.4*	*4.4*	*4.4*	*4.4*	*	*	*	*	*	*
2048	*4.4*	*4.4*	*4.4*	*4.4*	*	*	*	*	*	*
4096	*4.4*	*4.4*	*4.4*	*4.4*	*	*	*	*	*	*
8192	*4.4*	*4.4*	*4.4*	*4.4*	*	*	*	*	*	*
16384	*4.4*	*4.4*	*4.4*	*4.4*	*	*	*	*	*	*
32768	-	*4.4*	*4.4*	*4.4*	*4.4*	*	*	*	*	*
65536	-	-	*4.4*	*4.4*	*4.4*	*4.6*	*4.6*	*	*	*
131072	-	-	-	*4.3*	*4.3*	*4.3*	*4.6*	*	*	*
262144	-	-	-	-	*4.3*	*4.3*	*4.6*	*4.6*	*4.6*	*
524288	-	-	-	-	-	*4.3*	*4.6*	*4.6*	*4.6*	*4.6*
1048576	-	-	-	-	-	-	*4.6*	*4.6*	*4.6*	*4.6*
2097152	-	-	-	-	-	-	-	*4.6*	*4.6*	*4.6*
4194304	-	-	-	-	-	-	-	-	*4.6*	*4.6*
8388608	-	-	-	-	-	-	-	-	-	*4.6*

Table 4.2 Performance matrix of mixed mode chain tree broadcast on the IBM cluster(8x16 MPI tasks). The value in each entry means the best prediction equation. A * means all equations give prediction with error less than 5%.

	16K	*32K*	*64K*	*128K*	*256K*	*512k*	*1M*	*2M*	*4M*	*8M*
1024	*38%*	*17%*	*7.7%*	*0.7%*	*	*	*	*	*	*
2048	*38%*	*20%*	*9.1%*	*1.6%*	*	*	*	*	*	*
4096	*37%*	*22%*	*13%*	*2.8%*	*	*	*	*	*	*
8192	*29%*	*21%*	*15%*	*7.3%*	*	*	*	*	*	*
16384	*26%*	*20%*	*12%*	*5.0%*	*	*	*	*	*	*
32768	-	*10%*	*12%*	*8.2%*	*4.0%*	*	*	*	*	*
65536	-	-	*2.7%*	*1.1%*	*-7.1%*	*6.2%*	*-2.7%*	*	*	*
131072	-	-	-	*18%*	*-9.3%*	*-15%*	*-0.7%*	*	*	*
262144	-	-	-	-	*-2.9%*	*-20%*	*-1.5%*	*2.7%*	*-4.0%*	*
524288	-	-	-	-	-	*-10%*	*3.7%*	*3.7%*	*-5.7%*	*-0.4%*
1048576	-	-	-	-	-	-	*12.9%*	*-0.8%*	*-6.3%*	*-2.7%*
2097152	-	-	-	-	-	-	-	*-6.0%*	*-10%*	*-5.8%*
4194304	-	-	-	-	-	-	-	-	*-19%*	*-4.9%*
8388608	-	-	-	-	-	-	-	-	-	*-3.7%*

Table 4.3 Error of the performance equations in Table 4.2. A positive number indicates over estimate while a negative number means under estimate.

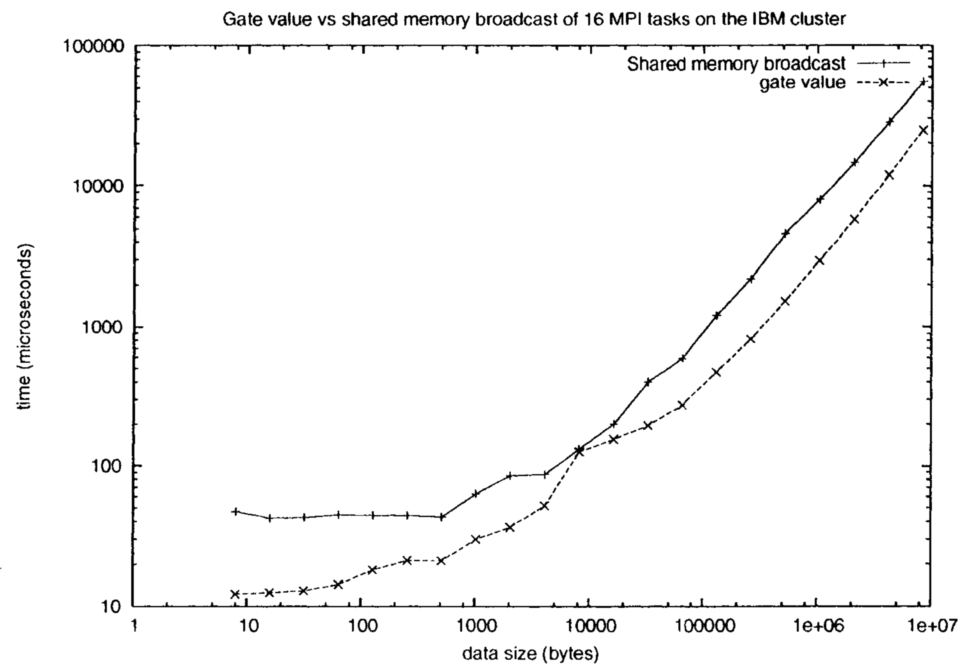


Figure 4.13 Gate values and the cost of shared memory broadcast (1x16 MPI tasks) on the IBM cluster.

Our model assumes that the cost of inter-node collective communications (less processes) should be less or at least equals to mixed mode collective communications (more processes). Theoretically this is true. On some occasions, however, we found that the performance of mixed mode collective communications are better than inter-node collective communications. For example, segment size 16KB in Figure 4.9, 8x1 MPI tasks using inter-node chain tree takes more time than 8x16 MPI tasks using mixed mode chain tree. We speculated this is due to MPI implementations. On the IBM cluster posting too many non-blocking calls of small messages may degrade the performance.

To verify this, we used the same implementation of inter-node chain tree broadcast for 8x1 MPI tasks, gradually add a certain amount of dummy computation between two non-blocking sends. When this dummy computation is large enough, we observed a performance improvement on the IBM cluster. It happens only when there is flood of non-blocking sends of small message segment size. This also explains why the complete overlapping equation, which is supposed to provide the lower bound, still over estimates the performance as shown in Table

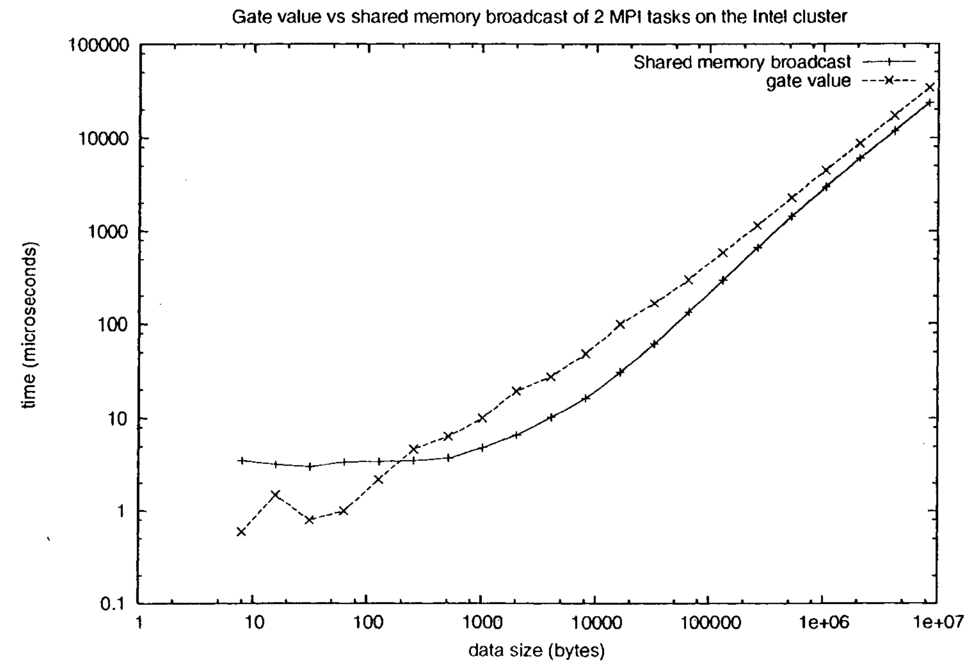


Figure 4.14 Gate value and the cost of shared memory broadcast (1x2 MPI tasks) on the Intel cluster.

4.3.

2. Given a {*segment size, overlapping size*} pair, it is difficult to determine the performance mode without testing.

We used three equations to describe the performance characteristics of mixed mode chain tree broadcast. It would be better if we could condense them into one equation. However, it is difficult to determine which equation provide the best prediction for a certain entry in the performance matrix, especially the entries in the penalty mode. Through experiments we found that the penalty mode is not just affected by memory bandwidth, but also by the number of segments, segment size, overlapping size, pipeline mechanisms. Identifying the best equation for an entry is almost as complex as tuning mixed mode collective communications.

4.5.5 More on the Overlapping Penalty

Through experiments, we also observed that the cost of processing the remaining segment $smcc(m_r)$ may also contribute to the overall overlapping penalty. In the chain tree case under

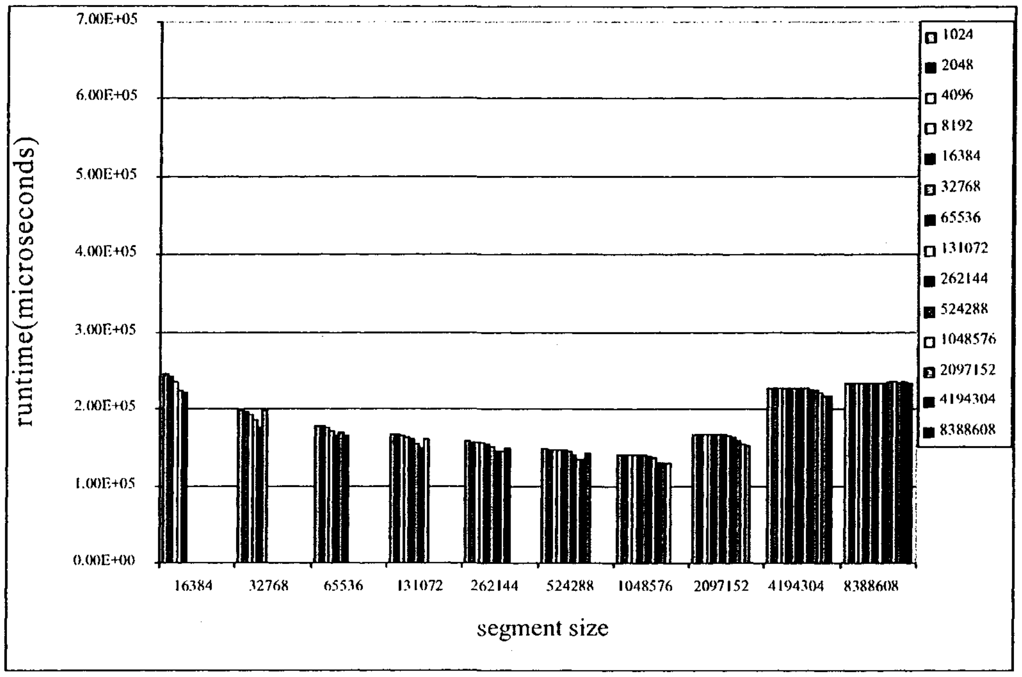


Figure 4.15 The predictions of overlapping mode (equation 4.3).

penalty mode, the root node posts non-blocking send for the **last** segment m_i, then processes shared memory broadcast of the last overlapping segment m_s. Right after *smcc(m_s)* time the root goes on processing the remaining segment m_r. It is possible that the MPI layer is not able to gain access to memory bus between two shared memory broadcasts. In that case the overlapping penalty is higher than the prediction based on equation 4.6.

The overall penalty for the overlapping mechanisms depends on algorithms, and is somewhat statistical. For example, for flat tree broadcast, we found that most of the time *smcc(m_r)* contributes to the overall overlapping penalty; for chain tree broadcast under penalty mode, equation 4.6 gives better predictions most of the time.

An implication from the overlapping penalty is that, when we are designing algorithms for overlapping communication and computation, we may want to avoid a non-blocking sends followed by a huge amount of memory access to retrieve data for computation. From our experimental results, this may in fact degrade the performance of communications.

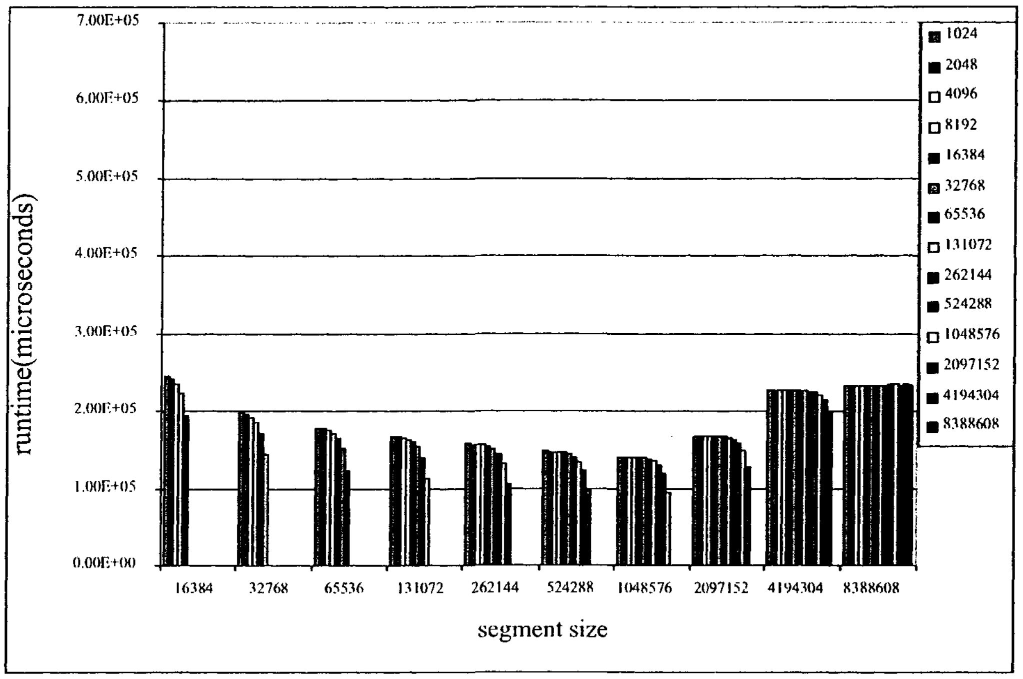


Figure 4.16 The predictions of complete overlapping mode (equation 4.4).

4.6 Tuning Strategies

Kielmann *et al.* [11] and Vadhiyar *et al.* [6] tuned the performance by directly apply basic values (*g(m), os(m), or(m), L*) on the performance equations of different collective algorithms then used the results from their predictions. For mixed mode collective communications, we can not take this direct estimate approach. As mentioned in the previous section, it is difficult to determine which performance equation to use. We usually have to examine a certain entry in the performance matrix in order to determine the best equation.

Instead of a direct estimate, we take an indirect approach: filter out unnecessary experiments. In this section we discuss several tuning strategies. The tuning strategies are based on heuristics; however, they provide much better tuning times and the quality of the tuned performance is very close to the results of the exhaustive tuning method. We use the performance matrix of mixed mode chain tree to broadcast 8MB message on the IBM cluster(8x16 MPI tasks) as an example to show how to filter out unnecessary experiments step by step.

1. Select the entries with large overall overlapping size. If the overall overlapping size is

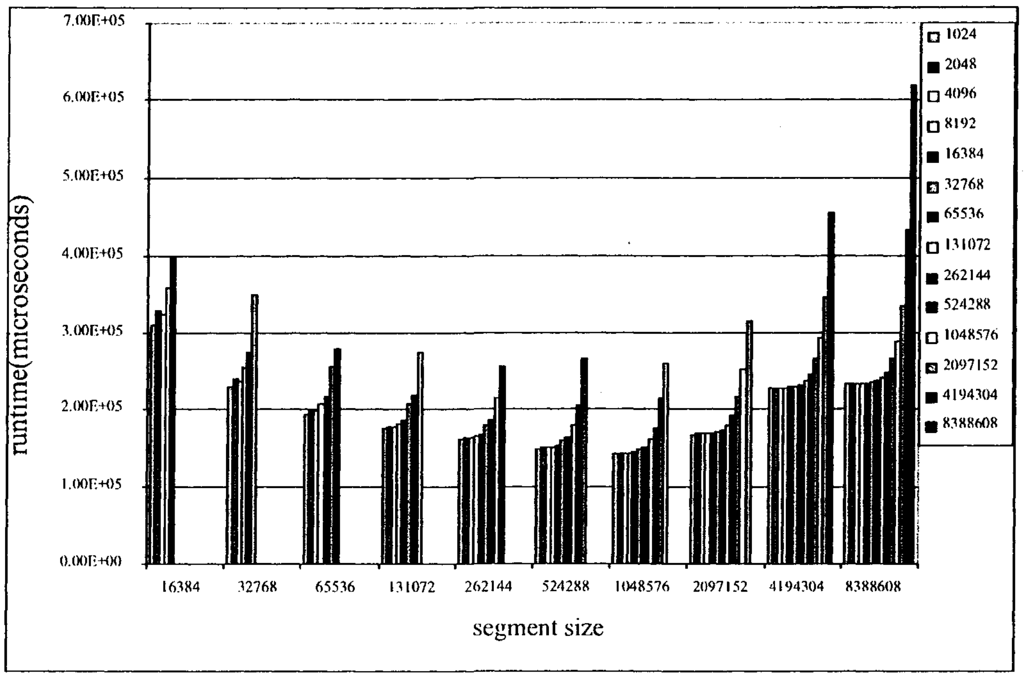


Figure 4.17 The predictions of penalty mode (equation 4.6).

too small, the performance improvement of overlapping may be very limited. In that case we may just select a non-overlapped approach. Our experimental results indicate that the overall overlapping size must be at least more than 25% of the total message size in order to gain the performance improvement. Based on this observation, for each column in the performance matrix, we select only the bottom three elements for testing. The possible entries to be examined after this filter is shown in table 4.4.

2. Reduce the range of experiments. The performance of the overlapping approach should not be worse than a non-overlapped approach. Thus the results of a mixed mode collective communication without overlapping provides the upper-bound as the acceptable performance. For an entry in the performance matrix, if the cost of its inter-node communication is already more than this upper-bound, we discard it. This is shown in Figure 4.18. Table 4.5 shows the possible entries to be examined after this filter.

3. Start testing from the smallest segment, it examines only the bottom entry of each column. Good performance can be achieved only without overlapping penalty (overlapping or complete overlapping mode), and the performance of mixed mode collective communications

	16K	32K	64K	128K	256K	512k	1M	2M	4M	8M
1024	-	-	-	-	-	-	-	-	-	-
2048	-	-	-	-	-	-	-	-	-	-
4096	x	-	-	-	-	-	-	-	-	-
8192	x	x	-	-	-	-	-	-	-	-
16384	x	x	x	-	-	-	-	-	-	-
32768	-	x	x	x	-	-	-	-	-	-
65536	-	-	x	x	x	-	-	-	-	-
131072	-	-	-	x	x	x	-	-	-	-
262144	-	-	-	-	x	x	x	-	-	-
524288	-	-	-	-	-	x	x	x	-	-
1048576	-	-	-	-	-	-	x	x	x	-
2097152	-	-	-	-	-	-	-	x	x	x
4194304	-	-	-	-	-	-	-	-	x	x
8388608	-	-	-	-	-	-	-	-	-	x

Table 4.4 Possible entries to examine for mixed mode chain tree broadcast of 8M after filter 1. x: entries to examine.

	16K	32K	64K	128K	256K	512k	1M	2M	4M	8M
1024	-	-	-	-	-	-	-	-	-	-
2048	-	-	-	-	-	-	-	-	-	-
4096	-	-	-	-	-	-	-	-	-	-
8192	-	-	-	-	-	-	-	-	-	-
16384	-	-	x	-	-	-	-	-	-	-
32768	-	-	x	x	-	-	-	-	-	-
65536	-	-	x	x	x	-	-	-	-	-
131072	-	-	-	x	x	x	-	-	-	-
262144	-	-	-	-	x	x	x	-	-	-
524288	-	-	-	-	-	x	x	x	-	-
1048576	-	-	-	-	-	-	x	x	-	-
2097152	-	-	-	-	-	-	-	x	-	-
4194304	-	-	-	-	-	-	-	-	-	-
8388608	-	-	-	-	-	-	-	-	-	-

Table 4.5 Possible entries to examine after filter 2.

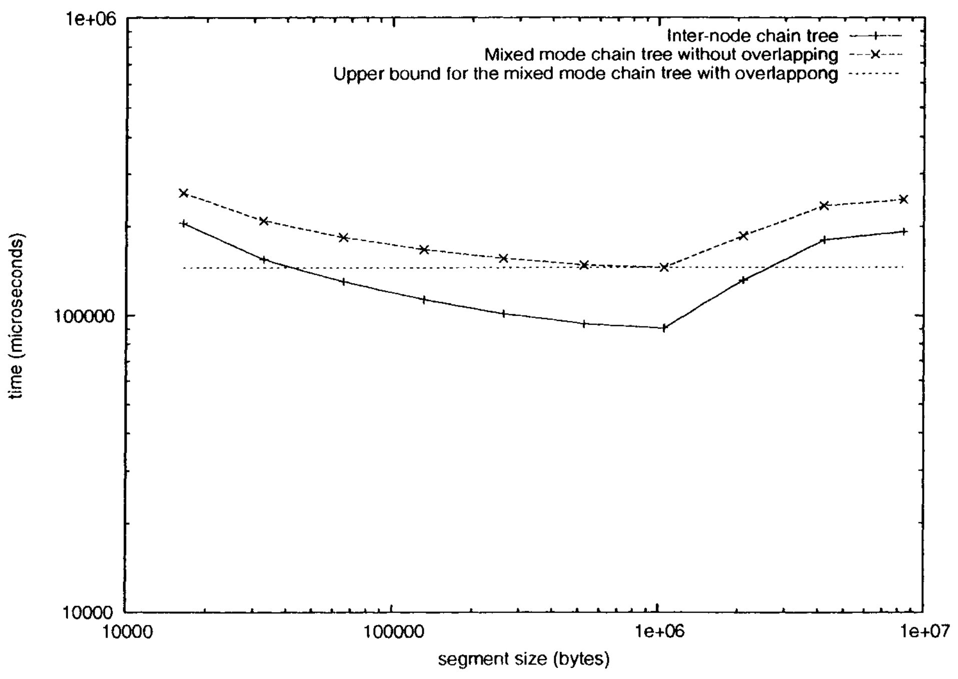


Figure 4.18 Finding the upper-bound and the experiment range (filter 2).

under overlapping mode is likely to improve with the increase of overlapping size, (see Figure 4.15 and 4.16). Thus after applying filter 1 and 2, we start testing from the bottom element of the smallest segment column. When the result of the entry is available, we compare it to the equations from the performance model. We can identify which performance equation gives the best prediction, then use this equation for the other two entries in the same column. If we find an entry *(i,j)* is under penalty mode, we do not process any more entries of lager segment size or overlapping size (for example, entries *(i+1, j+1)*). Table 4.6 shows the entries to be examined and entries to use prediction equations.

Table 4.7 and 4.8 show the results of filtered tuning time and the exhaustive tuning time to tune our implementations of three mixed mode broadcast algorithms, on the IBM cluster using 8x16 MPI tasks and on the Intel cluster using 8x2 MPI tasks. For each {*segment size, overlapping size*} we run it 10 times and then calculate the average. Figures 4.19 shows our worse tuning result of mixed mode chain tree on the IBM cluster. The performance loss by the filtered tuning is about 5% to 10% from 16KB to 64KB. For binomial tree and binary tree, our filtered tuning results are the same as exhaustive tuning results.

	16K	*32K*	*64K*	*128K*	*256K*	*512k*	*1M*	*2M*	*4M*	*8M*
1024	-	-	-	-	-	-	-	-	-	-
2048	-	-	-	-	-	-	-	-	-	-
4096	-	-	-	-	-	-	-	-	-	-
8192	-	-	-	-	-	-	-	-	-	-
16384	-	-	*c*	-	-	-	-	-	-	-
32768	-	-	*c*	*c*	-	-	-	-	-	-
65536	-	-	*X*	*c*	*c*	-	-	-	-	-
131072	-	-	-	*X*	*c*	*c*	-	-	-	-
262144	-	-	-	-	*X*	*c*	-	-	-	-
524288	-	-	-	-	-	*X*	-	-	-	-
1048576	-	-	-	-	-	-	-	-	-	-
2097152	-	-	-	-	-	-	-	-	-	-
4194304	-	-	-	-	-	-	-	-	-	-
8388608	-	-	-	-	-	-	-	-	-	-

Table 4.6 Possible entries to examine and calculate after filter 3. X: entries to examine. c: entries using prediction equations.

	exhaustively tuning	*filtered tuning*	*percentage*
chain tree	*358*	*39*	*10.8%*
binary tree	*379*	*38*	*10%*
binomial tree	*392*	*41*	*10.4%*

Table 4.7 Comparison of tuning time (seconds) on the IBM cluster, 8x16 MPI tasks, 16K to 8M

	exhaustively tuning	*filtered tuning*	*percentage*
chain tree	*229*	*17*	*7.4%*
binary tree	*267*	*22*	*8.2%*
binomial tree	*248*	*14*	*5.6%*

Table 4.8 Comparison of tuning time (seconds) on the Intel cluster, 8x2 MPI tasks, 4K to 8M.

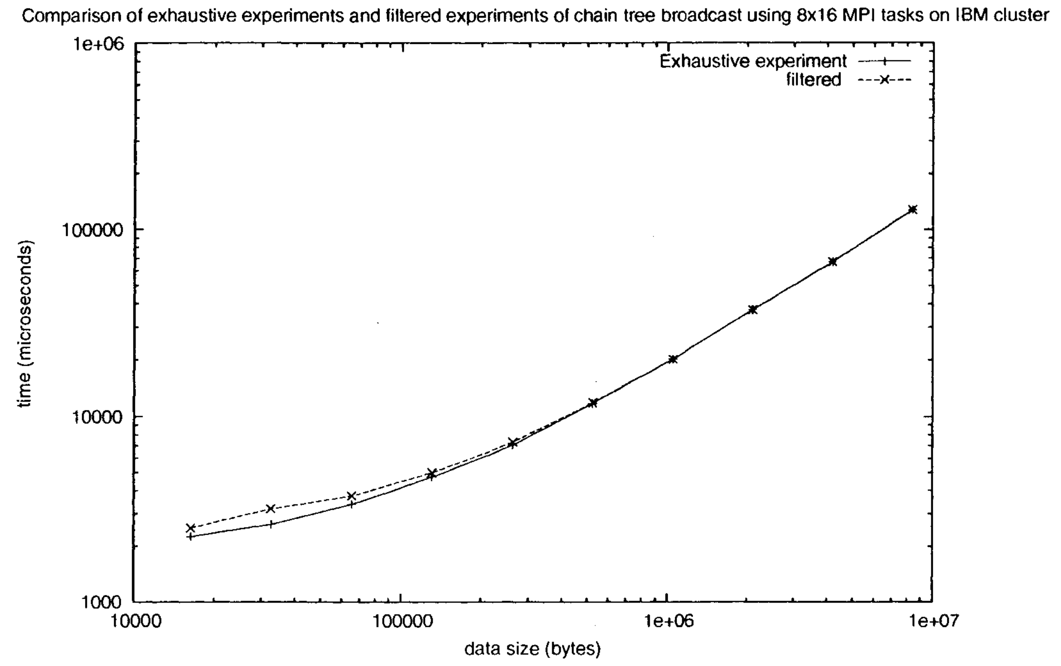


Figure 4.19 Performance comparison of exhaustive tuning and filtered tuning of mixed mode chain tree broadcast.

Even if we are only tuning for total overlapping (overlapping size equals segment size), we can still reduce significant amount of tuning time. For example, on the IBM cluster using 8x16 MPI tasks, to find the best setting for binary tree broadcast on 8MB message, exhaustive tuning takes more than 20 seconds (examine 12 entries). Using filter 2 and 3 we need only less than 6 seconds to find the optimal setting (examine 4 entries).

4.7 Summary

Mixed mode collective communications provide better performance over the other approaches on SMP clusters; however, without a good performance model, exhaustively testing every setting is the only strategy to tune the performance of mixed mode collective communications on different SMP platforms. It is not only time consuming but also impractical.

In this chapter we first explored the characteristics of mixed mode collective communications. We showed that the performance of mixed mode collective communications does not always improve or decrease with the increase of overlapping segment size. We then developed a performance model that is able to describe these characteristics. The model is different from the other performance models for collective communications: it considers both shared memory and point-to-point collective communications. In this model we also introduce "overlapping penalty", which can happen in the mixed mode collective communications. Based on the performance model and experimental results we developed several tuning strategies; our results show that the strategies can reduce the amount of tuning time up to less then 10% of exhaustive tuning. The performance model and tuning strategies outlined in this chapter have been accepted as a technical conference paper at Supercomputing 2005 [116].

In the next chapter we will discuss how to put every modules developed in this dissertation together as the foundation to build an automatic tuning system for collective communications on SMP clusters.

CHAPTER 5. MICRO-BENCHMARKS AND TUNING PROCEDURES

5.1 Introduction

In the previous chapters several new methods are developed for collective communications on SMP clusters; these methods include: portable optimizations using the SMP architecture, a performance model to predict the performance of mixed mode collective communications, and several tuning strategies. In this chapter we will discuss the procedures to build a practical automatic tuning system, and how to put the developed methods into the structure of such a system.

5.2 Tuning Points for a Parallel Application

Two existing automatic tuning systems for collective communications, MagPIe and ACCT, provide tuning strategies base on performance models, which in turn are based on several network parameters. MagPIe provides supporting tools [12] to evaluate gate value, send overhead and receive overhead on a target platform. ACCT does not provide such tools; the tuning parameters and tuning decisions must be hand coded then re-compile their programs on a target platform.

The two systems provide several good approaches to design automatic tuning systems. However, many tuning processes are left to a user. Consider the mixed mode collective communications we developed in the previous chapters. The tuning parameters include {*algorithms, implementations, segment size, overlapping size, shared memory buffer size, number of shared memory buffer, nodes, processors per node*}. If we provide only the tunable implementations of a collective communication to users, they have to find the optimal values for parameters by themselves. Without supporting tools for tuning, a user can only exhaustively test every

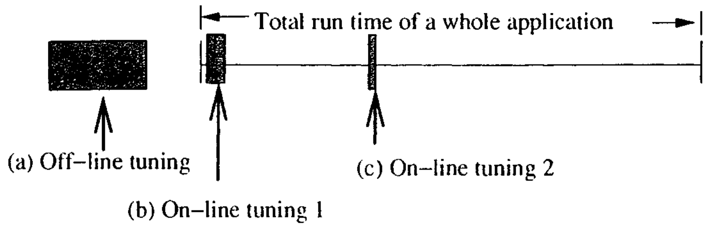


Figure 5.1 Possible tuning points, and relative allowed tuning time for collective communications.

possible configuration to find the optimal setting.

For a practical automatic tuning system, it should include tunable implementations of collective communications and tools to tune the performance for users. The tuning tools can be regarded as a set of micro-benchmarks for collective communications. The functionalities of micro-benchmarks include: gathering basic information of a target system, predicting performance of implementations base on the collected system information, filtering out unnecessary experiments and selecting a minimum set of implementations that requires runtime experiments.

To develop such a system, we first identify the "tuning points" during the execution of a parallel application where we can tune the performance of collective communications. There are three tuning stages during the execution of a parallel application where we can tune the parameters for the collective communication implementations.

Stage 1: off line tuning. Before an application starts; the micro-benchmarks can be used to evaluate basic information and select the implementations for runtime testing.

Stage 2: on-line tuning stage one. After an application is started, but before any computation or communication is executed. The run time environment is set and we can start the testing procedure to find the best implementation of a collective communication.

Stage 3: on-line tuning stage two. During the runtime of an application, the performance of collective communications or applications can be gathered, and adjustments can be made according to the interactions between collective communication library and applications.

Figure 5.1 shows possible tuning stages and relative tuning time for each stage in a parallel application. The micro-benchmarks for each tuning stage are different since the purpose and the acceptable tuning time are different from stage to stage. The tuning results from an earlier stage are provided as the base for the next tuning stage. We will discuss the three tuning stages in the following sections. The methods developed in this dissertation are for the tuning in the off-line tuning stage and the on-line tuning stage one.

5.3 Off-line Tuning - Tuning Before an Application Starts

At this stage, the system uses the micro-benchmarks to gather basic network information such as gate value, send overhead and receive overhead on the inter-node layer. On SMP clusters, the performance of shared memory operations and shared memory collective communications should also be gathered during this stage.

Since this is intended to be tested off-line, the tuning time can be a couple hours and use only a small set of nodes. The goal is to use only a few nodes, within limited amount of time to provide filtered information for the next tuning stage.

The micro-benchmarks in this stage can be divided into three smaller sets, mapped onto the three layers in our programming model. They are discussed in the following subsections.

5.3.1 Micro-benchmarks for the Intra-node Layer Collective Communications

The micro-benchmarks on this layer explore the characteristics of intra-node layer collective communications designed with the basic shared memory operations. Besides the performance of individual shared memory operations, we also need the performance of a shared memory collective communications since we use these values for performance predictions of mixed mode collective communications.

The micro-benchmarks on the intra-node layer should provide the following information:

(1). Overhead of intra-node communications through NIC and shared memory.

(2). Bandwidth of the intra-node layer communications.

(3). The best shared buffer size for shared memory operations.

(4). The best number of pipeline buffers.

(5). The cost of individual shared memory operations.

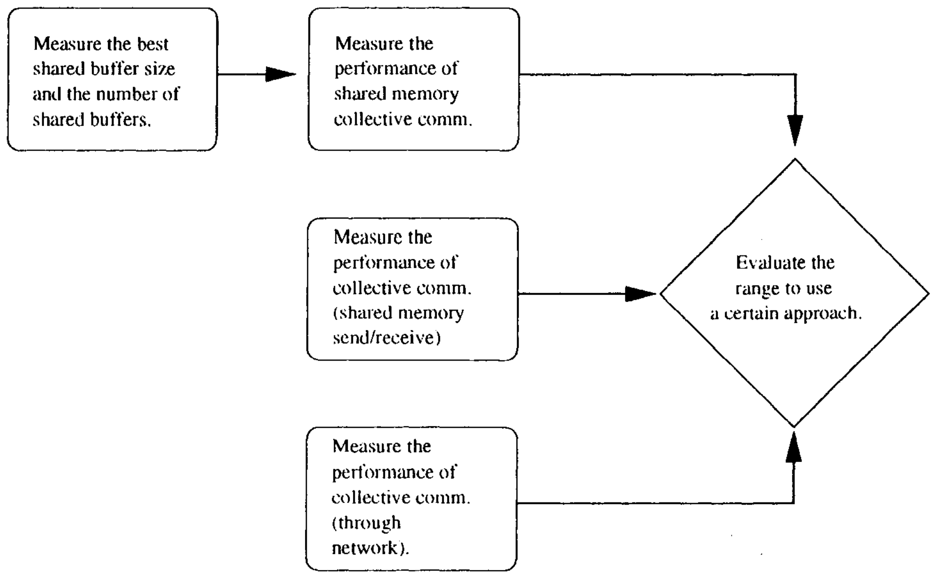

Measure the best shared buffer size and the number of shared buffers.
Measure the performance of shared memory collective comm.
Measure the performance of collective comm. (shared memory send/receive)
Evaluate the range to use a certain approach.
Measure the performance of collective comm. (through network).

Figure 5.2 Block diagram of the tuning procedure on the intra-node layer.

(6). The cost of shared memory collective communications.

(1) and (2) can be measured using the tools provided by MagPIe, (3) to (6) are developed in this research.

The block diagram of the tuning procedure for the intra-node layer is shown in Figure 5.2. We first need to find out what is the best size for a shared buffer, and the best number of shared buffers for pipelining; those methods are described in Chapter 2. When the two data are available, we can evaluate the cost of shared memory collective communications. If shared memory send and receive are available on the target systems, we can also measure the cost of inter-node collective communications algorithms using shared memory send and receive. We also need to measure the performance of collective communications using interconnection networks within an SMP node.

When all the results are available, we can decide the range to use a certain implementation for a collective operation. This part requires only one SMP node, and can be completed within a short time.

5.3.2 Micro-benchmarks for the Inter-node Layer Collective Communications

The methods to explore network characteristics on this layer are extensively explored in many literatures. For automatic tuning, several key information are required: the segment size for a pipelined implementation, the switching point between blocking and non-blocking implementations, the testing range for parameters of an implementation, and the performance predictions to filter out unnecessary experiments of the implementations that do not provide good performance.

Overall, on the inter-node layer, the micro-benchmarks should provide the following:

(1). Overhead, gate value, bandwidth and latency of the inter-node layer communications.

(2). Switching point between latency-bound and bandwidth-bound communications.

(3). Performance prediction of an inter-node layer collective communication algorithm base on information such as latency and overhead. This can filter out some unnecessary experiments.

(4). The proper range of segment size for run time testing. This is for pipelined implementations.

MagPIe provides the information for (1). For (3) we can use the tuning strategies in MagPIe or ACCT. These strategies can be designed as modules and incorporate them into the tuning system. The data for (2) and (4) can be developed base on the information provided by MagPIe tools.

The block diagram of the tuning procedure for the inter-node layer is shown in Figure 5.3. We first use the tool provided by MagPIe to extract the results of gate value, send overhead, receive overhead and latency on the target systems, then we calculate the performance of collective communications on the inter-node layer. For the latency bound communications, we use Hockney's model for performance prediction; for bandwidth bound communications, we use either P-LogP model or the performance model in ACCT. In this dissertation we implemented performance prediction routines for only a few collective operations; the performance prediction modules of ACCT and MagPIe are scheduled to release with OpenMPI and we expect to be able to use those modules without too much modifications.

When the performance predictions of collective communications and switching point be-

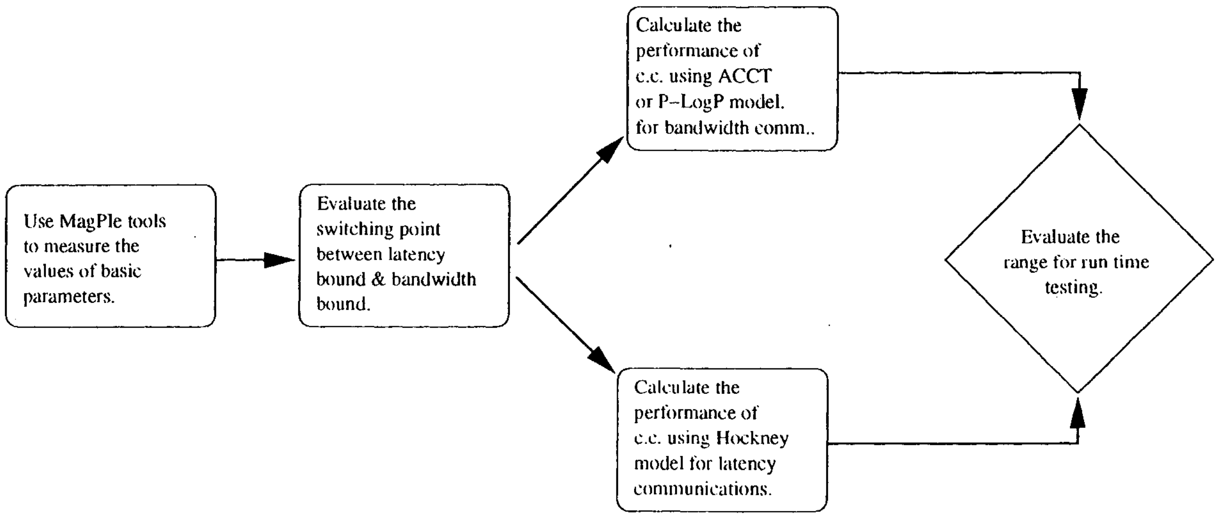

Use MagPle tools to measure the values of basic parameters.
Evaluate the switching point between latency bound & bandwidth bound.
Calculate the performance of c.c. using ACCT or P-LogP model. for bandwidth comm..
Calculate the performance of c.c. using Hockney model for latency communications.
Evaluate the range for run time testing.

Figure 5.3 Block diagram of the tuning procedure on the inter-node layer.

tween latency/bandwidth bound communications are available, we can set the range for experiments in the next tuning stage: run time tuning stage one.

5.3.3 Micro-benchmarks for the Inter-node/Intra-node Overlapping Mechanism

Exploring the cost of overlapping and predicting the performance of a mixed mode collective communication after overlapping inter-node/intra-node communications are the major functionalities of micro-benchmarks in this group.

(1). Overlapping penalty when overlapping layers of communications.

(2). The range for run time testing, given a {*segment size, overlapping size*} pair for a particular mixed mode collective communication.

(3). Performance prediction of a certain implementation of a mixed mode collective communication. For example, given the cost of a shared memory gather operation, a shared memory broadcast operation and an inter-node ring all-gather operation, we should be able to predict the performance of the upper-bound of a mixed mode all-gather implementation.

The tuning procedure is shown in Figure 5.4. We need the information from both inter-node and intra-node micro-benchmarks. First we need the latency of shared memory collective communications and inter-node collective communications to calculate the cost of mixed mode collective communications without overlapping, and we use this information to determine the testing points in the filter 3 as discussed in the previous chapter. We then examine these testing points, and calculate the other settings by using the results from runtime experiments, and find the setting that gives the best performance.

The tuning procedures discussed in this section is only a guideline; it shows how to put the methods developed in this dissertation into tuning procedures for an automatic tuning system. Many information discussed in the earlier chapters can be used for different purposes but they are not shown in our tuning procedures. For example, by measuring the cost of chain tree broadcast and compare it with binomial tree scatter, we can estimate that the performance of chain tree scatter can be very poor and is not worth implementing.

The information from off-line tuning should be of the table format, with each row indicating the testing range for an implementation. The first runtime tuning stage uses this table to fine tuning the performance before starts any computation or communication for an application.

5.4 Online Tuning One - Tuning Before Application Computation Starts

At this stage, the system should use the information gathered from off-line micro-benchmarks and runtime information such as number of processes, and possible message size, to find the proper implementations for a collective operation.

As an example, on IBM SP system using IBM's propriety network and running IBM's MPI, from an off-line micro-benchmark we can know that the possible switching point between immediate mode and rendezvous mode is 4K. This suggests the smallest segment size for blocking/non-blocking implementations is around 4K. Exactly which value is the best for an algorithm or an implementation has to be determined based on the number of processes given at runtime. At this stage we may want to test the best blocking implementation of a certain collective operation from 1K to 16K, doubling testing size each time, and then test a few non-blocking implementations using the same testing range. When the results are available we can find the exact switching point.

These experiments must be designed in a way that they can be completed in a very short

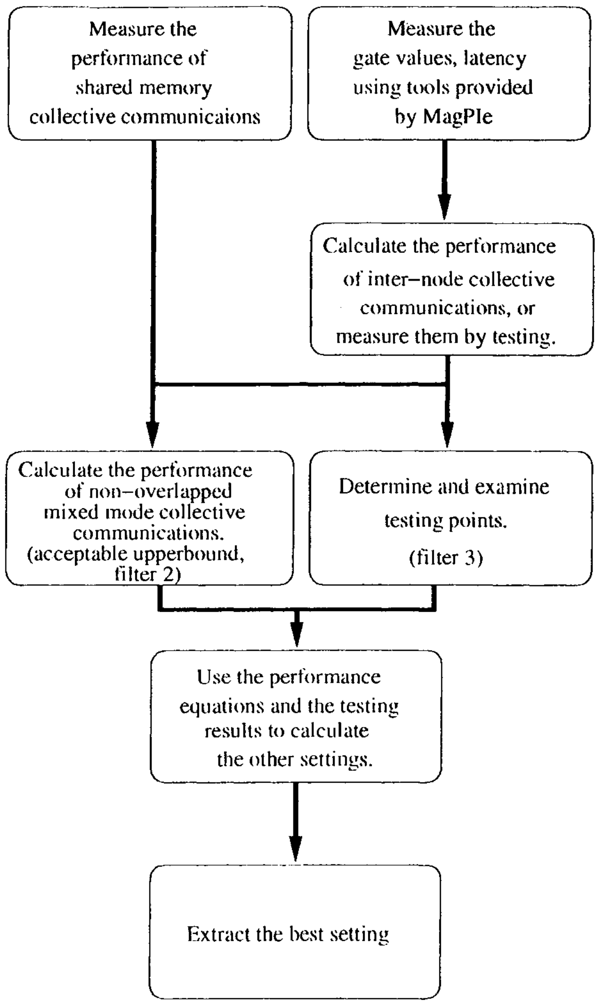

Measure the performance of shared memory collective communicaions
Measure the gate values, latency using tools provided by MagPIe
Calculate the performance of inter–node collective communications, or measure them by testing.
Calculate the performance of non–overlapped mixed mode collective communications. (acceptable upperbound, filter 2)
Determine and examine testing points. (filter 3)
Use the performance equations and the testing results to calculate the other settings.
Extract the best setting

Figure 5.4 Block diagram of the tuning procedure for the overlapping mechanism.

time, and completing all required experiments should be no more than a few minutes. We want the cost of this tuning to be subsumed into the overall runtime of a parallel application. Obviously, the tuning time in this stage depends on the quality of the off-line tuning stage.

One research topic that may be useful in this stage is topology discovery [61, 29, 30, 62]. While exploring topology is a research topic in GRID computing or distributed computing, our focus is a cluster of SMP nodes, and the difference of node-to-node latency within an SMP cluster may not be large enough to take topology into consideration unless the physical network is constructed in a certain topology. One such cluster is the SCINK cluster at SCL. We did not take topology into account in this research.

Also, it is desirable that the results of the runtime experiments can be stored in the system so in the future if another application is requesting the same configuration, the system can directly extract the required information without doing the same experiment for every application.

5.5 Runtime Tuning - Interacting with Parallel Applications

There are many research topics in this part such as gathering the run time performance data, setting the threshold for tuning, and interacting with applications and making runtime tuning decisions [57, 56, 59, 60, 58]. Some of those may impact the runtime performance of collative communications; for example, the computation load on a node may also affect the performance of its communications, as shown in the work by Sosonkina *et.al* [57, 56]. Incorporating this runtime information into our system can be one of our future research topics.

Active Harmony system [63, 64, 65, 66] is able to tune the performance of a system during runtime, and can handle a large number of tuning parameters. The basic assumption of Active Harmony is that the effect of each tuning parameter is independent to the overall performance; this is not the case in tuning collective communications. However, some of the designing methodologies such as exploring priority of each tuning parameter [66] can still be considered for future research in this system.

Designing an automatic tuning system that interacts with applications is a long-term research and requires cooperation of several research projects. To our best knowledge, there is no existing collective communication library with the capability to interact with applications during run time. This is one of our research topics in the future.

To sum up, the tuning strategies for our system are: 1. Use off-line tuning to minimize the runtime experiments, then 2. Use the results from runtime experiments to find the best setting.

If the quality of off-line tuning is good enough, the cost of run-time tuning can be subsumed into the runtime of parallel applications. This extends the idea of MagPIe into the run time

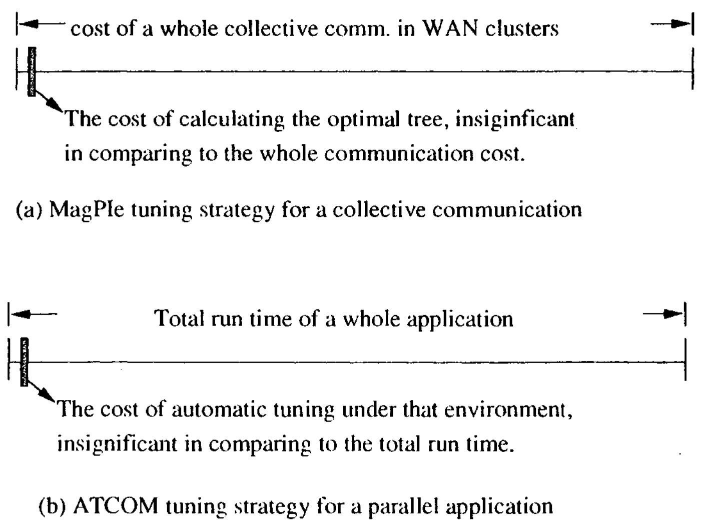

cost of a whole collective comm. in WAN clusters
The cost of calculating the optimal tree, insiginficant in comparing to the whole communication cost.
(a) MagPIe tuning strategy for a collective communication
Total run time of a whole application
The cost of automatic tuning under that environment, insignificant in comparing to the total run time.
(b) ATCOM tuning strategy for a parallel application

Figure 5.5 Tuning Strategies of MagPIe.

of a whole parallel application instead of a single collective communication, and is shown in figure 5.5.

5.6 Summary

To design a practical automatic tuning system, not only do we need tunable implementations of collective communications, but also tools that help users automatically tune for the optimal performance. This chapter discusses the micro-benchmarks for different tuning stages, and gives a guideline of how to put the developed methods in this dissertation into tuning procedures. The system integration for this system will be the future work for ATCOM system.

CHAPTER 6. CONCLUSION

Optimizing collective communications in a parallel machine is a challenging design task that requires the integration of modeling, system analysis, parameter tuning, and benchmarking methods plus a good optimization strategy. Because of the degree of complexity, automatic tuning system is highly desired, particularly for SMP clusters. Existing approaches are not designed for SMP clusters, and our experimental results have shown that their performance is not close to the optimal one.

In this research, we have built an infrastructure for designing an automatically tuned system for collective communications on SMP clusters. We first constructed a programming model for designing collective communications on SMP clusters. The programming model is based on a well understood communication model, which includes both point-to-point based communications and shared memory concurrent memory access feature. Using this model, on the intra-node layer we have successfully developed a shared memory collective communication library that utilizes the concurrent memory access feature of the SMP architecture. It provides significant performance improvement over the other point-to-point based collective communication libraries. We then constructed a generic mechanism to overlap inter-node and intra-node collective communications. These two components construct the generic programming model for designing collective communications on SMP clusters. Although the two methods have been studied in platform-specific optimizations, we have successfully adapted them into a platform-independent automatic tuning system. Its performance is significantly better than the platform-specific ones.

The mixed mode collective communications involve complex interactions between layers of communications, and the performance characteristics of these types of communications are very different from those of the point-to-point based ones. We explored these special characteristics, and developed a performance model that takes into account those interactions between communication layers. Several performance equations are required to describe the characteristics of a mixed mode collective communication, and the existing tuning strategies can not be directly applied to tune the performance of mixed mode collective communications. We have developed several tuning strategies that are based on heuristics. The experimental results showed that the tuning time can be reduced dramatically from the brute force exhaustive search while providing almost the same tuning quality. We have also showed how to organize the developed methods into a set of micro-benchmarks and procedures to form the building blocks of a practical tuning system.

While the optimizations outlined in this dissertation do not follow the exact definition of "message passing", it is still based on a model that can be applied to most SMP clusters. With proper programming techniques and collaborations with the MPI developers, it is possible to incorporate our library into the existing MPI implementations, and the existing MPI based applications can use this library without any modification.

Automatic tuning communication systems are vital to the performance of time consuming, communication intensive applications. A computation intensive application may benefit very little, or none at all, from an automatic tuning communication system. For example, in a parallel matrix multiplication, each computation stage requires $O(n^3)$ of computation time while each communication stage requires only $O(n^2)$ communication time. The communication time contributes only a small fraction of the total application run time thus it does not demand optimization of communications. A fast and communication intensive application may also benefit very little from the tuning system since the tuning time may exceed the total run time of the application. Our rationale in using an automatic tuning communication system is that, for a time consuming, communication intensive application that runs days, weeks or even months, it needs to use only a few minutes for performance tuning and the overall performance can be improved. The approaches of the existing tuning systems are only for single processors clusters. With the optimized collective communications, the new performance model and tuning strategies, we have extended automatic tuning communication systems to work on multi-processor clusters.

There are several directions for the future work of this research. A possible short-term project is to extend the current library to include more collective operations and explore more tuning strategies. While our tuning strategies show very promising tuning time and quality on our testing platforms, we want to verify our approaches on more platforms and explore more tuning strategies for different kinds of SMP clusters.

One possible long-term project is to introduce an interacting mechanism between the tuning system and the applications. As mentioned in the last chapter, this may require cooperation between several research projects. Another possible long-term project is to enhance our programming model and performance model for different architectures. For example, if there is a processor dedicated to communications in every SMP node, how should we change our performance model? Will there still be an overlapping penalty? Do we need a different programming model to utilize this architecture?

With more and more complex parallel architectures available, designing automatic tuning systems for the parallel computers becomes even more difficult. Currently, we are seeing two opposing trends. Researchers are seeking approaches to reduce the programming complexity and burden in designing parallel applications. System architectures are becoming more complex. The trend of current CPU architectures, multi-core CPUs, will become the main stream of high performance computing hardware. Designing efficient parallel applications on this complex architecture is still an open problem.

One approach to design efficient parallel applications is to re-design the applications on new architectures. This approach is time consuming as well as error prone. It is quite possible that, by the time the new design is completed, there would be another new architecture available.

Another approach, which we have utilized, is to provide another layer of abstraction, and design tunable approaches and thus performance enhancement on this layer. With more complex memory hierarchies and interconnection technologies appearing in new parallel computers, designing these "virtual architectures" is also a research topic. On one hand, this virtual architecture should be detailed enough so the developers can use it to design more efficient applications; on the other hand it should not be too detailed as to inhibit portability.

The research in this dissertation is based on a more detailed "virtual architecture" for SMP clusters than the point-to-point based one used by existing message passing libraries, and the results have shown portability and performance improvement can be achieved on different SMP clusters. The results are encouraging for the design of tunable applications on future large-scale, complex parallel computers.

APPENDIX MIXED MODE MATRIX MULTIPLICATION

Introduction

In modern clustering environments where the memory hierarchy has many layers (distributed memory, shared memory layer, cache, ...), an important question is how to fully utilize all available resources and identify the most dominant layer in certain computation. When combining algorithms on all layers together, what would be the best method to get the best performance out of all the resources we have? Mixed mode programming model that uses thread programming on the shared memory layer and message passing programming on the distributed memory layer is a method that many researchers are using to utilize the memory resources. In the research in this appendix, we take an algorithmic approach that uses matrix multiplication as a tool to show how cache algorithms affect the performance of both shared memory and distributed memory algorithms. We show that with good underlying cache algorithm, overall performance is stable. When underlying cache algorithm is bad, super-linear speedup may occur, and increasing number of threads may also improve performance.

The content of this appendix was published in 2002 IEEE Cluster Computing Conference. From this project we can observe the complex procedure to re-design a parallel algorithm with mixed mode programming. We investigated this approach before conducting the research on improving MPI collective communications.

Memory Hierarchies in the modern clustering environments

Figure A.1 shows the memory hierarchy that exists in most nodes of modern clustering environments. Globally, many nodes are linked together by a high-speed network; inside each

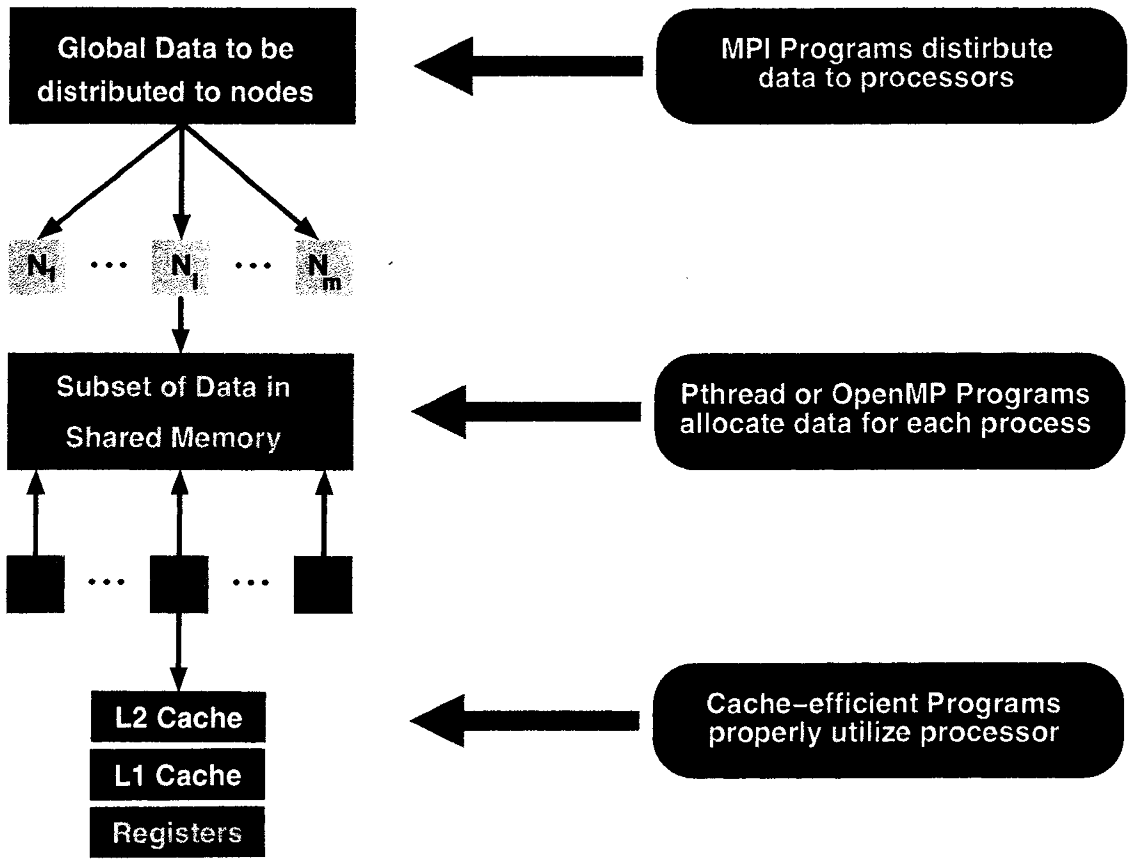

Global Data to be distributed to nodes
MPI Programs distirbute data to processors
N1 ... Ni ... Nm
Subset of Data in Shared Memory
Pthread or OpenMP Programs allocate data for each process
L2 Cache
L1 Cache
Registers
Cache–efficient Programs properly utilize processor

Figure A.1 Mixed mode programming model

node there may be many processors; along with each processor memory access is either to a high speed memory unit "cache" or the low speed "main memory".

In our mixed-mode programming model we use message passing interface, MPI, for the data communication between the global nodes. Inside each MPI process we have two choices, one is to use POSIX threads for creating threads, one or many threads may belong to the MPI process mapped to the node. The other choice is again to use MPI for local processes mapped to all processors of the node. Inside each process we use different algorithms that utilize the cache. Another option for threads that was not explored in this project was to use the OpenMP standard. Based on the specific programming model, we selected several matrix multiplication algorithms on each layer and implemented them.

Bova et. al., [97] determined that, "On a 100-CPU machine, using 100 MPI workers to perform a 100-component harbor simulation is inefficient due to inappropriate load balance.

It would be more efficient to have 25 MPI workers create four OpenMP threads for each assigned wave component." In our experiments, we show that even in a perfectly load balanced computation such as matrix multiplication, the overall mixed mode performance is highly affected by cache algorithms.

Our testing platform is the IBM SP system at the National Energy Research Scientific Computing facility [77]. It is a distributed-memory parallel supercomputer with 184 compute nodes. Each node has 16 POWER3+ processors and at least 16 GBytes of memory, thus at least 1 GBytes of memory per processor. Each node has 64 KBytes of L1 data cache and 8192 Kbytes of L2 cache. The nodes are connected to each other with the IBM proprietary switching network.

Cache layer matrix multiplication algorithms

The cache based algorithms used in our research vary from those that have high cache misses to those that effectively use cache. This is by no means a complete coverage of all possible cache algorithms but is representative of those used and taught in the community. Also there is no performance optimization beyond the definition of each algorithm such as what is done in the ATLAS suite [98] where an optimal implementation is produce by balancing tradeoffs between operation count, memory access patterns, etc,. and computed performance metrics. Different optimization techniques can also be found in Crawford and Wadleigh [99] or Dowd and Severance [100].

Simple three loops algorithm

Figure A.2(a), simple three loops algorithm: the figure shows the memory access pattern of this algorithm. This algorithm will incur the most cache misses among all other cache algorithms introduced here. However, it is the algorithm with fewest instruction count. LaMarca and Ladner [101] as well as Chatterjee et. al. [102], mention that for a cache algorithm to get the best performance, the recursion should terminate whenever a block of data fits into cache and then call the algorithm with fewest instruction count. In our implementation, whenever

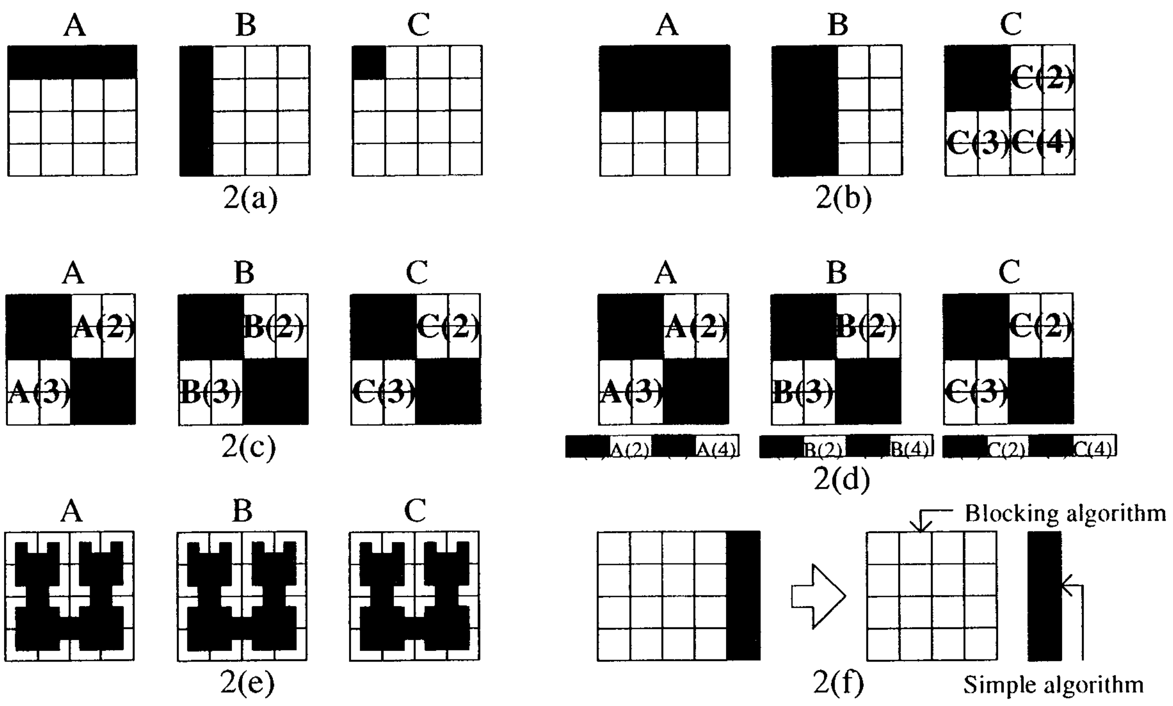


Figure A.2 Pictorial Representation of Cache Level Algorithms.

data blocks fit completely in cache, this algorithm is used.

Blocking C algorithm

Figure A.2(b), compute matrix C block by block. Since the algorithm computes according to the square patch of C, the corresponding portions of matrices A and B are not required to be square. As the matrix dimensions increase, the size of A and B patches also increase and thus incur more cache misses. This algorithm performs well when the sizes of matrices are small. If matrix B is transposed first in order to access elements in B consecutively, the performance is much better. The results presented in this work do not transpose matrix B.

Blocking A algorithm

Figure A.2(c), fix a block of A or B: the figure outlines the memory access order of this algorithm. The algorithm keeps a square patch of matrix A in the cache as long as possible. For example, block A(1) is computed with block B(1) and result put into C(1), then A(1) is

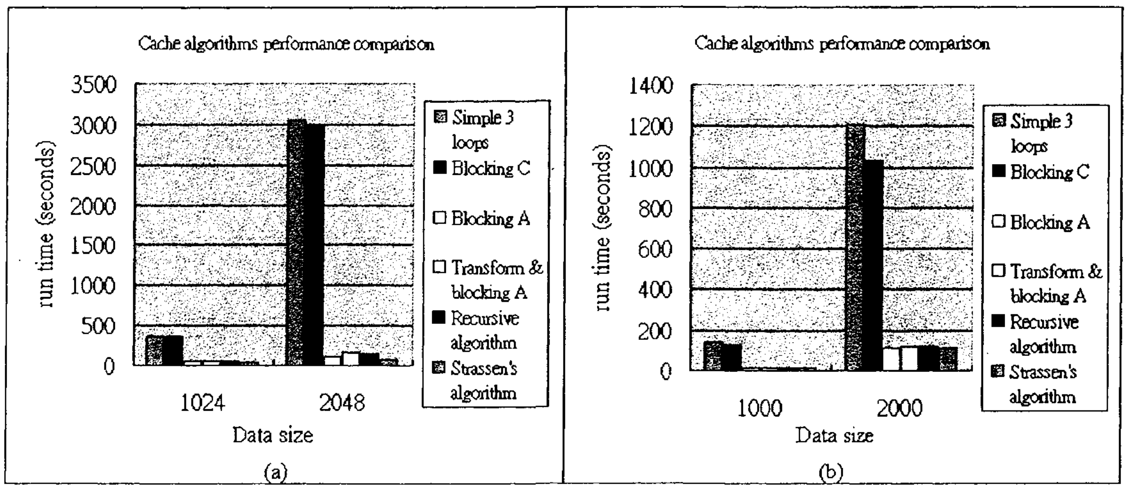


Figure A.3 Results of Cache Based Algorithms.

multiplied by B(2) and stores the data in C(2). A(1) is then swapped out and replaced by A(2) to multiply B(3) and B(4), and so on.

Transform and blocking A algorithm

Figure A.2(d), transform then blocking A: The memory partitioning scheme and computational order for this cache algorithm is the same as the previous algorithm. The only difference is that the layout of all three matrices are transformed before computation starts. The layout of elements in each block is made consecutive by creating a block that is small enough to fit into cache and copying the appropriate portion to the newly allocated block.

Recursive algorithm

Figure A.2(e), recursive layout: Chatterjee and coworkers [102, 103] and Frens and Wise [104] describe the recursive layout of matrix multiplication that the data is transformed into layout according to different space-filling curve order [105]; then computations are done recursively according to that order. Because of the recursion, data has to be a power of 2. Different methods of Chatterjee et. al., [102, 103] and Frens and Wise [104] are used to handle the case when data size is not a power of 2. We implemented a simple version of the "U layout" which

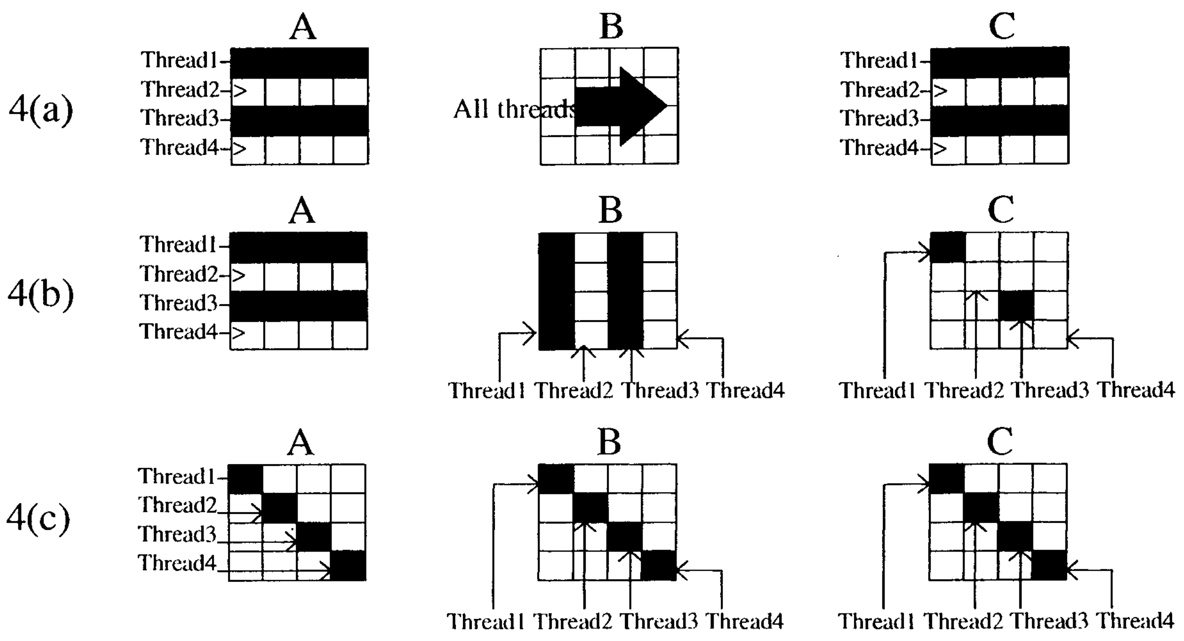


Figure A.4 Pictorial Representation of Shared Memory Algorithms

works on only square matrices. The blocking shell takes care of the case when data size are not a power of 2.

Strassen's algorithm

In theory Strassen's algorithm [106, 107] has better run time for matrix multiplication; it use additions and subtractions to reduce the times for multiplication. The algorithm processes data in small blocks recursively, which make it implicitly cache efficient.

Strassen's algorithm, cache oblivious algorithm [108] or Dag-consist algorithm [109] have the advantage that the programs do not need a threshold parameter to adjust the block size. In our experiments we found that recursion down to a single element reduced the performance and terminating the recursion at even a small block size increased the overall performance. For all of our implementations, we assign a cache size for each block of 10 Kbytes, which is available for most computers at this point in time.

Result and performance analysis of cache algorithms

Algorithms such as Strassen's algorithm work on square matrices only, and the dimension has to be a power of 2 to do recursion. When the dimension is not a power of two or the shape of matrices are not square, we use another blocking shell to compute the matrices with square blocks, and leave the rest parts that are not square to simple 3 loops algorithm. Figure A.2(f) shows this blocking shell method. In our implementation, three algorithms use this block shell - Strassen's algorithm, recursive algorithm and transform and blocking A algorithm.

Our results are shown in Figures A.3. In Figure A.3(a), when the matrix size is a power of 2, Strassen's algorithm shows the best performance. However, in a more realistic situation when we have to deal with the dimensions that are not a power of 2, or the shapes of matrices are not square, the cache misses caused by the block shell method and copying overhead reduces the performance of three algorithms - transform and blocking A, recursive algorithm and Strassen's algorithm. The best performing algorithm is to blocking A algorithm.

Programming issues

One issue to keep in mind is that, the result here is not to show that one algorithm is absolutely better than the others. It just means in our implementation, one algorithm shows better performance than the other. During our research we determined that programming cache algorithms is a nebulous task. The same algorithm be implemented in different ways giving a totally different performance. We use these algorithms, coded directly as describe above, simply as a basis to combine with algorithms in multiple layers of the memory hierarchy. We do *not* give any conclusion about which algorithm is optimal for the cache layer.

Shared memory layer matrix multiplication algorithms

We outline four possible shared memory matrix multiplication algorithms that can be easily implemented in the Pthreads programming model, represented in Figure A.4. This is by no means an exhaustive set of shared memory algorithms but representative of those that are used by the community.

Overlapping matrix B

Figure A.4(a), overlapping matrix B: the algorithm divides matrix A into several rectangle blocks horizontally according to the number of threads. Each thread computes certain rectangle block of matrix A with whole matrix B and produces a complete contribution to a portion of matrix C. The algorithm has the possibility of causing read contention when different threads try to read matrix B.

Non-overlapping algorithm

Figure A.4(b), non-overlapping: the algorithm divides matrix A horizontally and matrix B vertically into blocks. Each thread first computes a block of matrix A multiplied by a block matrix B thus produces a full contribution of a square block of matrix C. In the next stage every thread still use the same block of matrix A, but shifts to another block of matrix B, thus producing another full contribution to a different square block of matrix C. In this way, if all threads are executed concurrently then different threads have less chance of accessing the elements of matrix B at the same time when the number of threads is less than the number of blocks of A and B pairs.

Blocking algorithm

Figure A.4(c), blocking: the algorithm divides all three matrices into smaller square blocks, and each thread computes a square block of matrix A with a square block of B thus producing a partial square contribution to matrix C. The block computation order is the same as element computation order in simple 3-loop algorithm. Care must be taken to update matrix C atomically with mutex locks or assign the contributions required for the full block of C being computed to a single thread.

Transform and blocking algorithm

Transform and blocking: the algorithm has the same computation order and work on the same shapes of matrices as the previous algorithm, except before the computation begins, all

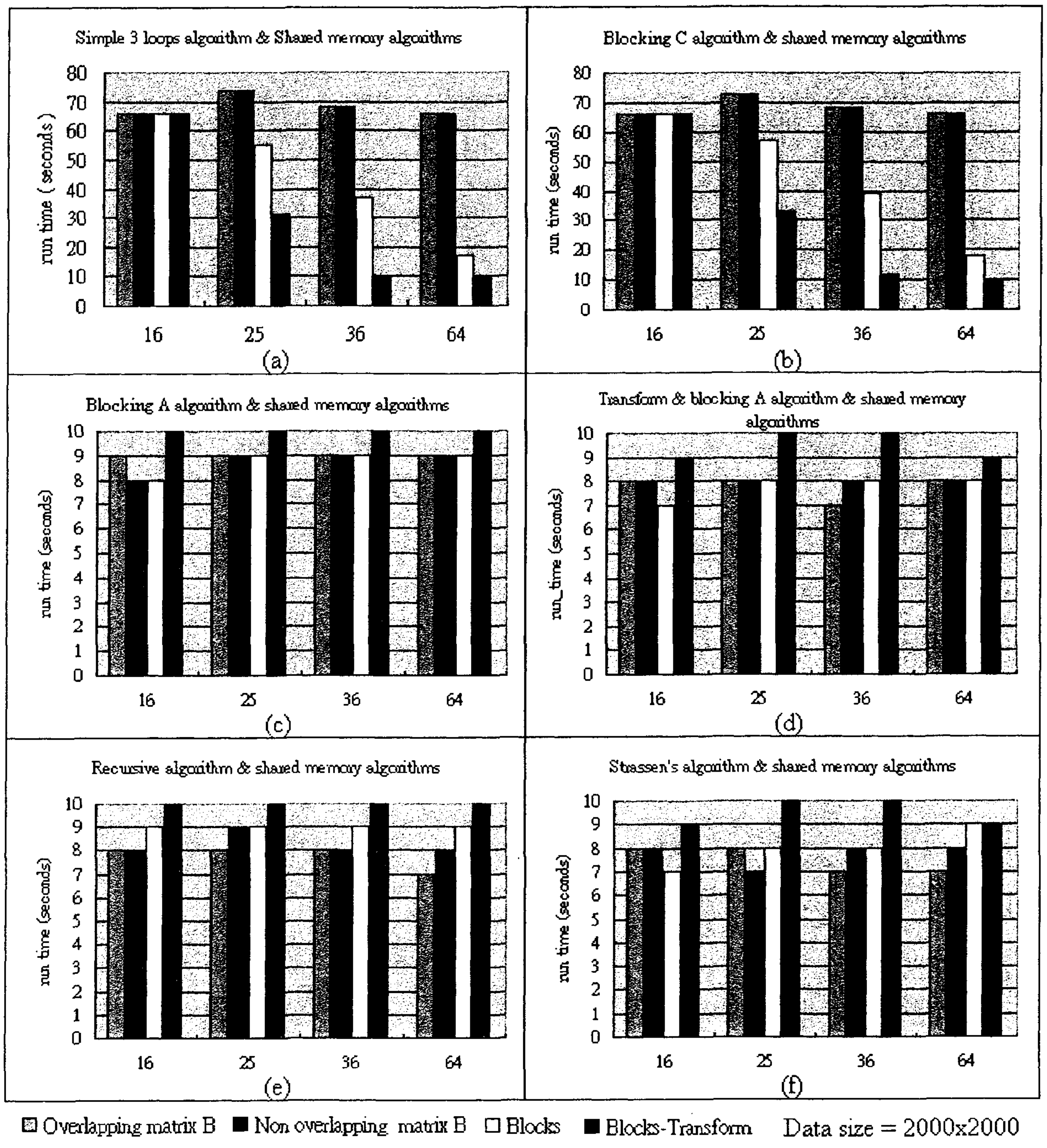


Figure A.5 Results of Shared Memory Based Algorithms

three matrices are transformed into blocks of consecutive data as in the cache algorithm delineated in section 6, and each thread works on three square patches with consecutive elements.

Results and performance analysis of shared memory algorithms

Our results are presented in Figure A.5(a) to A.5(f). Figure A.5(c) to Figure A.5(f) show that, with a good underlying cache algorithm, the performance of all four shared memory algorithms are similar. An increase in the number of threads does not substantially affect the overall performance.

On the other hand, Figure A.5(a) and figure A.5(b) show that, a "bad" cache algorithm combined with a shared memory algorithm that is insensitive to cache, has poor performance as the number of threads is increased. However, when a cache sensitive shared memory algorithm is combined with a "bad" cache algorithm, there is a performance gain with an increased number of threads. The performance can almost meet that of a good cache algorithm. This is due to the fact that as we increase the number of threads the smaller block size per thread actually fits into cache thus reducing the cache misses.

Distributed memory layer matrix multiplication algorithms

Here we focus on two common distributed matrix multiplication algorithms. Many more are available but these represent a common denominator of many algorithms.

Broadcasting algorithm

Two dimensional broadcasting algorithm: According to the number of physical nodes and physical grid, if the physical grid is $p \times q$, then matrices are partitioned into least common multiples of p and q parts. Each node takes turns broadcast the part of matrix A vertically or B horizontally or both, and computes according to the data it receives.

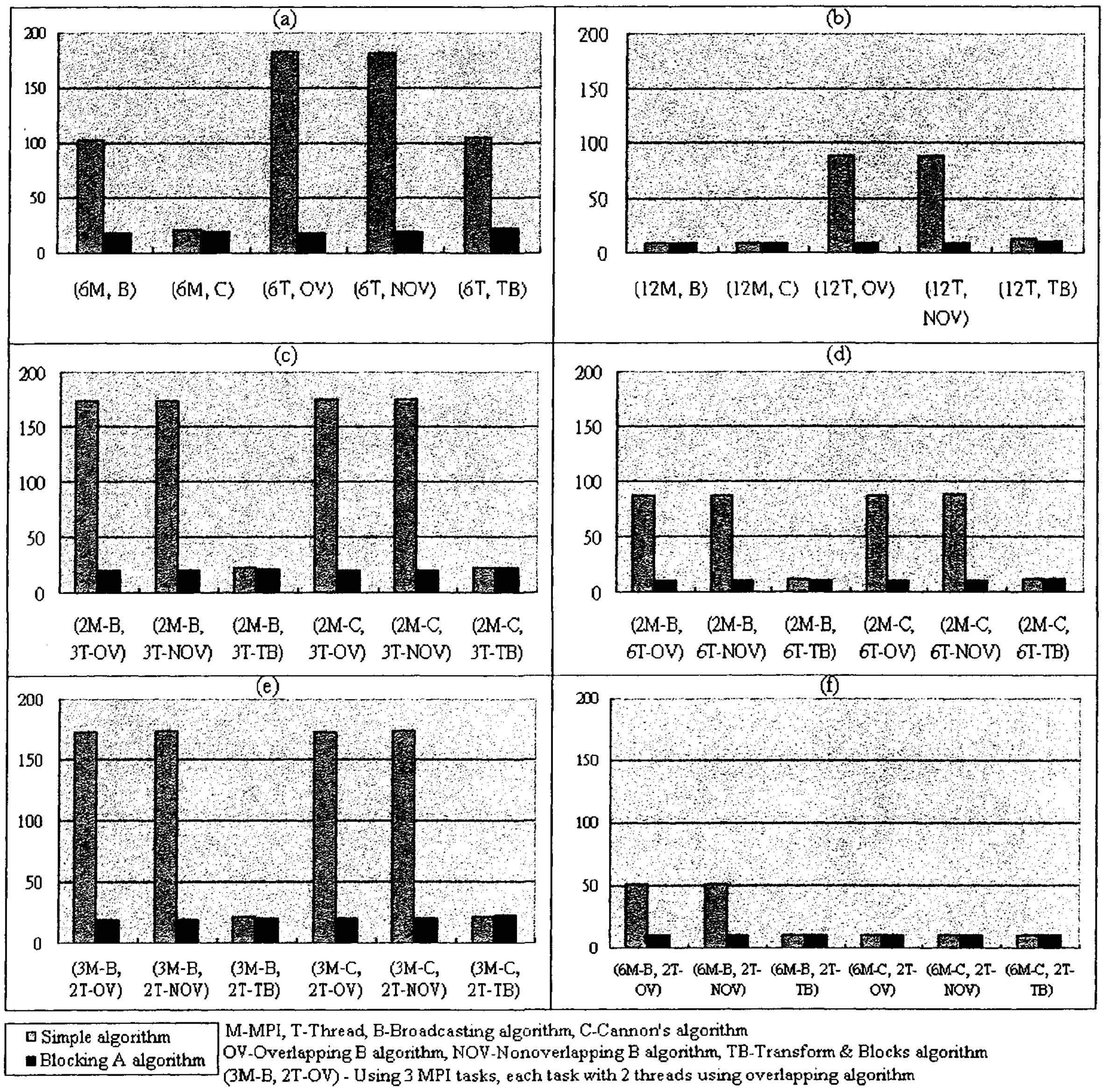


Figure A.6 Results of Mixed Mode Algorithms

Cannon's algorithm

The algorithm is described in almost every parallel algorithm book such as Kumar et. al., [110]. We use a generalized Cannon's algorithm similar to Lee and Fortes [111] instead of the original algorithm that requires the number of processors to form a certain square. In our implementation, we first form a grid using the available processors, then depending on the shape of the grid, we distribute data accordingly. If the grid is square, we distribute data as in Cannon's original algorithm. If the grid is rectangular, find the least common multiple of two dimensions, use the least common multiple as the dimension of a virtual square grid and then distribute data according to this virtual grid.

Result and performance analysis of Distributed Memory and Mixed Algorithms

In our observations, when using the same number of processors, MPI algorithms exhibit better performance than Pthreads algorithms when the underlying cache algorithm is "bad." The reason is that the distributed memory MPI algorithms always partition data into smaller blocks while shared memory algorithms work on a bulk data, and thus causing more cache misses.

However, with good underlying cache algorithm, the choice of how many MPI tasks combined with how many threads does not seem to be so important. Figure A.6(a) to A.6(f) show our result of computing matrices of size 2000x2000 using 6 and 12 nodes, with different combinations of algorithms, MPI tasks and thread tasks.

Figure A.6(a) and A.6(b) show that when bad cache algorithm combines with distributed algorithms or bad cache algorithm combines with shared memory algorithms, run time is not stable and doubling number of processors sometimes has super-linear speedup. Figure A.6(c) and A.6(d) show mixed algorithms of three layers and doubling number of processors have a speedup of 2. Figure A.6(e) and A.6(f) show mixed algorithms together and doubling number of processors when super-linear speedup happen again. With good underlying cache algorithm, performance are stable.

From Figure A.6(a) to A.6(f), we observed that, doubling the number of processors, good

	1000	2000	4000	8000	16000
Performance gain from improving MPI algorithm	0.43	0.244	1.348	8	12
Performance gain from improving cache algorithm	0.07	0.251	114	1018	> 6000

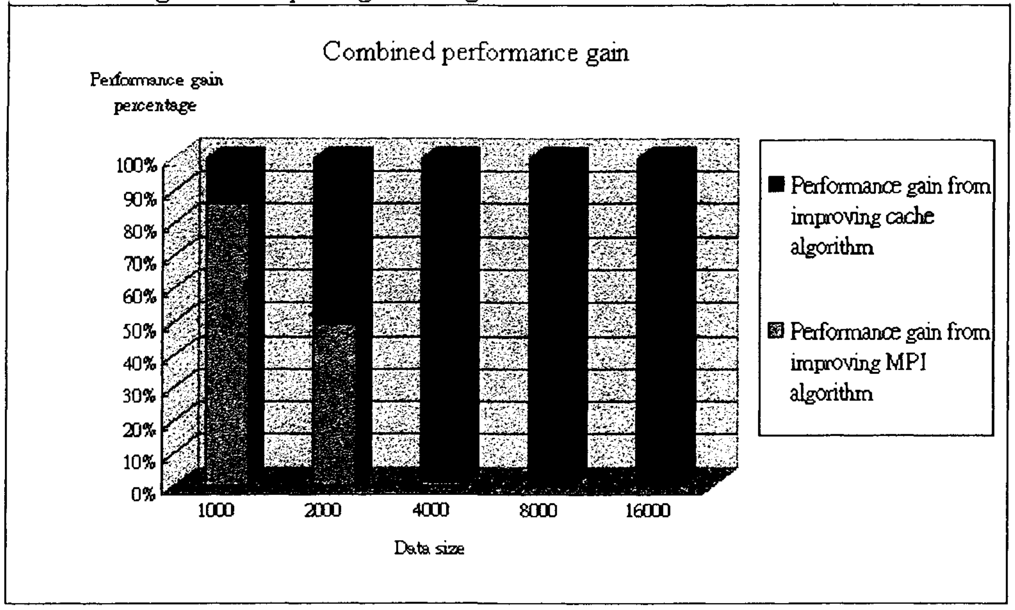


Figure A.7 Percentage of Performance Gain from Cache Layer and Distributed Memory Layer

cache algorithms give speedup of two while bad cache algorithms combines with algorithms from other layers that do not partition data small enough, also give speed up of two. On the other hand, performance of bad underling cache algorithms have super-linear speedup when data is partitioned small enough by algorithms from the other two layers.

These results show that, without a good cache algorithm, timings fluctuate. Different combinations of MPI tasks and numbers of threads exhibit different performance. The main dependence is on how the algorithms divide data and thus make good use of cache as the chunk size decreases. On the other hand, if the underlying cache algorithms is "good," the way data is partitioned matters much less.

We use a simple model to show the effects of cache algorithms:

$$\begin{aligned} T_{Total} &= T_{Comp} + \\ & \quad T_{Comm} + \end{aligned}$$

$$T_{Penalty}$$

where T_{Total} is the total time, T_{Comp} is the computational time, T_{Comm} is the communication time, and $T_{Penalty}$ is the time associated with cache misses. $T_{Comp} + T_{Comm}$ is the normal "total time" for many parallel computational models as described in Kumar et. al., [110], and for $T_{Comp} + T_{Penalty}$ model we refer to the two layers model of Matteo et. al., [108]. For matrix multiplication, communication cost is at most $O(n^2)$ while cache misses range from $O(n^2)$ to $O(n^3)$ depending on the algorithms. When data size are small, we can almost ignore cache misses penalty; when data size increases, cache misses penalty becomes a factor that affects total run time. When the data size is huge, $O(n^3)$ cache misses is now the bottleneck for the performance.

Figure A.7 shows the fraction of improvement coming from modifying the underlying cache algorithm for the distributed memory algorithms. We measured the total performance gain and the percentage that distributed layer algorithms and cache layer algorithms contribute. The data size are from 1000 to 16000, using 64 nodes. Shared memory algorithms are not used here since we don't have a node of 64 processors. As shown in figure A.7, when data size are small, all data block can fit into cache, the major improvement is from good distributed algorithms that reduce communication time. When data size increase, data block size also increase, incurring more cache misses, and cache algorithms became dominate contributer of performance gain. Eventually, cache layer algorithms contribute almost all performance gain when data size are very large.

Conclusion

In this paper we use different matrix multiplication algorithms on different layers to show how performance will be affected in mixed mode programming without a good cache algorithm, even when the work load is perfectly balanced. Since the core of parallel computations are still sequential computations, to improve the overall performance, not only do we need a model to utilize memory on every layer, but also good sequential core algorithms to achieve high performance. From our experiments, we believe that cache algorithms play a dominate role in many high performance computations, especially when processing large segments of data. If the computations is divided into many stages, and each stages only works on small data size, improving distributed algorithms improve the performance since cache misses do not matter much on computing small data size. On the other hand, if the computation has to work on large chucks of data, it is important to combine a good cache algorithms with an increase in the number of processors. Furthermore, parallel algorithms with "bad" underlying cache algorithms utilized in a mixed programming mode, the saturation of the thread space beyond the total number of computing threads equal to the number of available processors should provide a modest performance enhancement.

BIBLIOGRAPHY

[1] Roger W. Hockney. The Communication Challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing,* 20(3): 389–398. March 1994.

[2] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM,* 33(8): 103–111, 1990.

[3] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. *Proceedings of the Forth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* pages: 1–12, May 1993.

[4] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser and Chris Scheiman. LogGP: Incorporating Long Messages into the LogP Model One step closer towards a realistic model for parallel computation. *Journal of Parallel and Distributed Computing,* 44(1):71–79, 1997.

[5] Susanne E. Hambrusch and Ashfaq A. Khokhar. C^3: A Parallel Model for Coarse-grained Machines. *Journal on Parallel and Distributed Computing,* (32)2: 139–154, 1996.

[6] , Sathish S. Vadhiyar, Graham E. Fagg and Jack J. Dongarra. Toward an Accurate Model for Collective Communications. *International Journal of High Performance Computing Applications,* 18(1): 159–166, 2004.

[7] Csaba Andras Moritz, Matthew and I. Frank. LoGPC: Modeling Network Contention in Message-Passing Programs. *IEEE Transactions on Parallel and Distributed Systems,* 12(4): 404–415, April 2001.

[8] Steven Sistare, Rolf van de Vaart and Eugene Loh. Optimization of MPI Collectives Clusters of Large-Scale SMP's. *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM).*

[9] Vinod Tipparaju, Jarek Nieplocha and Dhabaleswar K. Panda. Fast Collective Operations Using Shared and Remote Memory Access Protocols on Clusters. *Proceedings of the 17th International Symposium on Parallel and Distributed Processing,* page: 84, 2003.

[10] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat and Raoul A. F. Bhoedjang. MAGPIE: MPI's Collective Communication Operations for Clustered Wide Area Systems. *Proceedings of ACM Symposium on Principles and Practice of Parallel Programming (PPoPP),* pages: 131–140, 1999.

[11] Thilo Kielmann, Henri E. Bal and Sergei Gorlatch. Bandwidth-efficient Collective Communication for Clustered Wide Area Systems. *Proceedings of the 14th International Symposium on Parallel and Distributed Processing,* page: 492, 2000.

[12] Thilo Kielmann, Henri E. Bal and Kees Verstoep. Fast Measurement of LogP Parameters for Message Passing Platforms. *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing,* pages: 1176–1183, 2000.

[13] Sathish S. Vadhiyar, Graham E. Fagg and Jack Dongarra. Automatically Tuned Collective Communications. *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM).*

[14] Graham E. Fagg, Sathish S. Vadhiyar and Jack J. Dongarra. ACCT: Automatic Collective Communications Tuning. *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface,* pages: 354–362, 2000.

[15] Rajeev Thakur and William Gropp. Improving the Performance of Collective Operations in MPICH. *Proceedings of the 10th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface.*

[16] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of Collective Communication Operations in MPICH. *Intrnational Journal of High Performance Computing Applications,* (19)1: 49-66, Spring 2005.

[17] Rolf Rabenseifner. Optimization of Collective Reduction Operations. *International Conference on Computational Science 2004,* pages: 1-9, 2004.

[18] Maciej Golebiewski, Rolf Hempel and Jesper Larsson Träff. Algorithms for Collective Communication Operations on SMP Clusters. *The 1999 Workshop on Cluster-Based Computing held in conjunction with 13th ACM-SIGARCH International Conference on Supercomputing.*

[19] Lars Paul Huse. Collective Communication on Dedicated Clusters of Workstations. *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface,* pages: 469–476, 1999.

[20] Prasenjit Mitra, David Payne, Lance Shuler, Robert van de Geijn and Jerrell Watts. Fast Collective Communication Libraries, Please. *Proceedings of the Intel Supercomputing User's Group Meeting,* 1995.

[21] Mike Barnett, Satya Gupta, David G. Payne, Lance Shuler, Robert van de Geijn, and Jerrell Watts. Interprocessor Collective Communication Library (InterCom). *Proceedings of the IEEE Scalable High Performance Computing Conference,* pages: 357–364, May 1994.

[22] Mike Barnett, Satya Gupta, David G. Payne, Lance Shuler, Robert van de Geijn, and Jerrell Watts. Building a HighPerformance Collective Communication Library. *Proceedings of the 1994 ACM/IEEE conference on Supercomputing,* pages: 107–116, 1994.

[23] Ernie Chan, Marcel F. Heimlich, Avi Purkayastha, and Robert van de Geijn. On Optimizing Collective Communication. *IEEE International Conference on Cluster Computing 2004.*

[24] Vasanth Bala, Jehoshua Bruck, Robert Cypher, Pablo Elustondo, Alex Ho, Ching-Tien Ho, Shlomo Kipnis and Marc Snir. A Portable and Tunable Collective Communication Library for Scalable Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems,* 6(2):154–164, February 1995.

[25] Dhabaleswar K. Panda. Issues in Designing Efficient and Practical Algorithms for Collective Communication on Wormhole-Routed Systems. *ICPP Workshop on Challenges for Parallel Processing,* pages: 8–15, August 1995.

[26] Joachim Worringen. Pipelining and Overlapping for MPI Collective Operations. *28th Annual IEEE International Conference on Local Computer Networks (LCN'03),* page: 548, 2003.

[27] Philip K. McKinley, Yih-jia Tsai and David F. Robinson. A Survey of Collective Communication in Wormhole-Routed Massively Parallel Computers. *Technical Report MSU-CPS-94-35, Michigan State University,* 1994.

[28] William Gropp, Ewing Lusk, Nathan Doss and Anthony Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing,* 22(6): 789–828, 1996.

[29] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC),* 63(5): 551–563, 2003.

[30] Ian Foster and Nicholas T. Karonis. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM),* pages: 1–11, 1998.

[31] Lars Paul Huse. MPI Optimization for SMP Based Clusters Interconnected with SCI. *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface,* pages: 56–63, 2000.

[32] Boris V.Protopopov and Anthony Skjellum. Shared-Memory Communication Approaches for an MPI Message-Passing Library. *Concurrency- Practice and Experience.* 12(9): 799–820, 2000.

[33] Toshiyuki Takahashi, Francis O'Carroll, Hiroshi Tezuka, Atsushi Hori, Shinji Sumimoto, Hiroshi Harada, Yutaka Ishikawa and Peter H. Beckman. Implementation and Evaluation of MPI on an SMP Cluster. *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing,* pages: 1178 - 1192, 1999.

[34] Thomas Worsch, Ralf H. Reussner and Werner Augustin. On Benchmarking Collective Operations. *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface,* pages: 271 - 279, 2002.

[35] William Gropp and Ewing Lusk. Reproducible Measurements of MPI Performance Characteristics. *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface,* pages: 11–18, 1999.

[36] William Gropp and Ewing Lusk. A High-Performance MPI Implementation on A Shared-Memory Vector Supercomputer. *Parallel Computing.* 22(11): 1513–1526, 1997.

[37] Natawut Nupairoj and Lionel M. Ni. Performance Metrics and Measurement Techniques of Collective Communication Services. *Proceedings of the First International Workshop on Communication and Architectural Support for Network-Based Parallel Computing,* pages: 212–226, 1997.

[38] Rolf Rabenseifner(1999). Automatic MPI Counter Profiling of All Users: First results on a CRAY T3E. *Proceedings of the Message Passing Interface Developer's and User's Conference 1999 (MPIDC'99),* pages: 77-85, 1999.

[39] Christian Bell, Dan Bonachea, Yannick Cote, Jason Duell, Paul Hargrove, Parry Husbands, Costin Iancu, Michael Welcome and Katherine A. Yelick. An Evaluation of Current High-Performance Networks. *Proceedings of the 17th International Symposium on Parallel and Distributed Processing,* page: 28, 2003.

[40] Fabrizio Petrini, Salvador Coll, Wu-chun Feng, and Adolfy Hoisie. Hardware- and Software-Based Collective Communication on the Quadrics Network. *IEEE International Symposium on Network Computing and Applications 2001 (NCA 2001),* page: 24, 2001.

[41] Fabrizio Petrini, Wu-chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network (QsNet): High-Performance Clustering Technology. *IEEE Micro,* 22(1): 46–57, 2002.

[42] Fabrizio Petrini, Eitan Frachtenberg, Adolfy Hoisie and Salvador Coll. Performance Evaluation of the Quadrics Interconnection Network. *Cluster Computing,* 6(2): 125–142, 2003.

[43] Introduction to InfiniBand. *A white paper for Mellanox Corporation.*

[44] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. *Proceedings of the 17th annual international conference on Supercomputing,* pages: 295–304, 2003.

[45] R. Noronha and Dhabaleswar K. Panda. Designing High Performance DSM Systems using InfiniBand: Opportunities, Challenges, and Experiences. *OSU Technical Report,* OSU-CISRC-11/03-TR60, 2003.

[46] Sushmitha P. Kini, Jiuxing Liu, Jiesheng Wu, Pete Wyckoff and Dhabaleswar K. Panda. Fast and Scalable Barrier using RDMA and Multicast Mechanisms for InfiniBand-Based Clusters. *Proceedings of the 10th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface,* 2003.

[47] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic and Wen-King Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro,* 15(1): 29–36, 1995.

[48] The GM Message Passing Systems. *Myricom Corporation, http: //www.myri.com/,* 1999.

[49] Darius Buntinas, Dhabaleswar K. Panda and P. Sadayappan. Fast NIC-Based Barrier over Myrinet/GM. *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS-01),* page: 52, 2001.

[50] Darius Buntinas, Dhabaleswar K. Panda and W. Gropp. NIC-Based Atomic Remote Memory Operations in Myrinet/GM. *Workshop on Nove Uses of System Area Networks(SAN-01), in conjunction with HPCA-8,* 2002.

[51] Darius Buntinas and Dhabaleswar K. Panda. NIC-Based Reduction in Myrinet Clusters: Is It Beneficial? *Workshop on Nove Uses of System Area Networks(SAN-02), in conjunction with HPCA-9,* 2003.

[52] Darius Buntinas, Dhabaleswar K. Panda and Ron Brightwell. Application-Bypass Broadcast in MPICH over GM. *Proceedings of The Third IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid),* 2003.

[53] Gigabite Ethernet Alliance. 10 Gigabit Ethernet - Technology Overview White Paper, 2002.

[54] Howard Frazier and Howard Johnson. Gigabit ethernet: From 100 to 1,000 Mbps. *IEEE Internet Computing,* 3(1): 24–31, 1999.

[55] Pavan Balaji, Piyush Shivam, Pete Wyckoff and Dhabaleswar K. Panda. High Performance User-Level Sockets over Gigabit Ethernet, *Proceedings of the IEEE International Conference on Cluster Computing,* page: 179, 2002.

[56] Sam Storie and Masha Sosonkina. Packet Probing as Network Load Detection for Scientific Applications at Run-time. *18th International Parallel and Distributed Processing Symposium (IPDPS'04),* page: 62b, 2004.

[57] Devdatta Kulkarni and Masha Sosonkina. A Framework for Integrating Network Information into Distributed Iterative Solution of Sparse Linear Systems. *High Performance Computing for Computational Science - VECPAR 2002, 5th International Conference, Porto, Portugal,* pages: 436–450, 2002.

[58] Rich Wolski. Dynamically Forecasting Network Performance Using the Network Weather Service. *Journal of Cluster Computing,* 1(1):119–132, 1998.

[59] Bruce B. Lowekamp. Combining active and passive network measurements to build scalable monitoring systems on the grid. *ACM SIGMETRICS Performance Evaluation Review,* 30(4): 19–26, 2003.

[60] Marcia Zangrilli and Bruce B. Lowekamp. Comparing Passive Network Monitoring of Grid Application Traffic with Active Probes. *Proceedings of the 4th International Workshop on Grid Computing (GRID2003),* page: 84, 2003.

[61] Nicholas T. Karonis, Bronis R. de Supinkski, Ian Foster, William Gropp, Ewing Lusk, and John Bresnahan. Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance. *Proceedings of the 14th International Symposium on Parallel and Distributed Processing,* page: 377, 2000.

[62] Bruce Lowekamp, David O'Hallaron and Thomas Gross(2001). Topology discovery for large ethernet networks. *ACM SIGCOMM Computer Communication Review,* 33(4): 237-248, 2001.

[63] Jeffrey K. Hollingsworth and Peter Keleher. Prediction and Adaptation in Active Harmony. *Cluster Computing,* 2(3):195–205, 1999.

[64] Cristian Tapus, I-Hsin Chung, Jeffrey K. Hollingsworth. Active Harmony: Towards Automated Performance Tuning. *Proceedings of the 2002 ACM/IEEE conference on Supercomputing,* pages: 1–11, 2002.

[65] I-Hsin Chung and Jeffrey K. Hollingsworth. Automated Cluster-Based Web Service Performance Tuning. *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04),* pages: 36–44, 2004.

[66] I-Hsin Chung and Jeffrey K. Hollingsworth. Using Information from Prior Runs to Improve Automated Tuning Systems. *Proceedings of the 2004 ACM/IEEE conference on Supercomputing,* page: 30, 2004.

[67] Message Passing Interface Forum. (1997). MPI-2: Extensions to the Message-Passing Interface. *http://www.mpi-forum.org/docs/docs.html.*

[68] MPICH2 Web page. *http://www-unix.mcs.anl.gov/mpi/mpich2/*

[69] High Performance Fortran Web Page. *http://dacnet.rice.edu/Depts/CRPC/HPFF/*

[70] OpenMP Web page. *http://www.openmp.org/drupal/*

[71] Virtual Interface(VI) Developer Forum. *http://www.vidf.org.*

[72] Automatically Tuned Linear Algebra Software (ATLAS) Web Page. *http://math-atlas.sourceforge.net/*

[73] OpenMPI Web page. *http://www.open-mpi.org/*

[74] LAM MPI Web page. *http://www.lam-mpi.org/*

[75] Top 500 Supercomputer list Web page. *http://www.top500.org/*

[76] Ralf Reussner. Special Karlsruher MPI - Benchmark. *http://liinwww.ira.uka.de/ skampi/.*

[77] NERSC web site. *"http://hpcf.nersc.gov/computers/SP.*

[78] Jeffrey S. Vetter and Frank Mueller. Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures. *Proceedings of the 16th International Symposium on Parallel and Distributed Processing,* page: 27, 2002.

[79] Jeffrey S. Vetter and Andy Yoo. An Empirical Performance Evaluation of Scalable Scientific Applications. *Proceedings of the 2002 ACM/IEEE conference on Supercomputing,* pages:1-18, 2002.

[80] Jarek Nieplocha and Bryan Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems. *Proceeding of 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) of International Parallel Processing Symposium IPPS/SPDP,* 1999.

[81] Jarek Nieplocha, V. Tipparaju, A. Saify, D.K. Panda. Protocols and Strategies for Optimizing Performance of Remote Memory Operations on Clusters. *Communication Architecture for Clusters (CAC'02) Workshop, held in conjunction with IPDPS '02,* 2002.

[82] Edgar Gabriel, Graham E Fagg, Antonin Bukovsky, Thara Angskun, and Jack Dongarra. A Fault-Tolerant Communication Library for Grid Environments. *17th Annual ACM International Conference on Supercomputing (ICS'03) International Workshop on Grid Computing and e-Science,* 2003.

[83] Steve Lumetta, Alan Mainwaring and David Culler. Multi-Protocol Active Messages on a Cluster of SMP's. *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM),* pages: 1-22, 1997.

[84] Edmond Chow and David Hysom. Assessing Performance of Hybrid MPI/OpenMP Programs on SMP Clusters. *Lawrence Livermore National Laboratory technical report,* UCRL-JC-143957, 2001.

[85] Franck Cappello, O. Richard and Daniel Etiemble. Investigating the performance of two programming models for clusters of SMP PCs. *Proceedings of the 6 th IEEE Symposium On High-Performance Computer Architecture (HPCA-6),* pages: 349-359, 2000.

[86] Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM),* 2000.

[87] David A. Bader and Joseph JáJá. SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors. *Journal of Parallel and Distributed Computing,* 58(1): 92–108, 1999.

[88] G.H. Shah, J.ieplocha, J. Mirza, C. Kim and R.K. Govindaraju. LAPI: A Low Level Communication Interface on the IBM RS/6000 SP: Experience and Performance Evaluation. *Cluster Computing,* 6: 115-124, 2003.

[89] Jack Dongarra, Ian Foster, Georffrey Fox, William Gropp, Ken Kennedy, Linda Torczeon, and Andy White. Sourcebook of Parallel Computing. *Morgan Kaufmann Publishers,* 2004.

[90] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake and C. V. Packer. BEOWULF: A Parallel Workstation for Scientific Computation. *Proceedings of the 24th International Conference on Parallel Processing,* 1: 11-14, 1995.

[91] T. Sterling, J. Salmon, D. J. Becker and D. Savarese. How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters. *MIT Press,* 1999.

[92] Al Geist, Adam Beguelin, Jack Dongarra and Weicheng Jiang. PVM:Parallel Virtual Machine. *The MIT Press,* 1994.

[93] H. Dachsel, J. Nieplocha and RJ Harrison. An Out-of-Core Implementation of the COLUMBUS Massively Parallel Multireference Configuration Interaction Program. *Proceedings of the 1998 ACM/IEEE conference on Supercomputing,* 1998.

[94] Cray Research, Inc. SHMEM Technical Note for C, SG-2516 2.3, October 1994.

[95] Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrell. Pthreads Programming: A POSIX Standard for Better Multiprocessing. *O'Reilly,* 1996.

[96] T. von Eicken, A. Basu, B. Buch and W. Vogels. U-Net: A User-level Network Interface for PArallel and Distributed Computing. *Proceedings of the 15th ACM Symposium on Operating Systems Principles(SOSP),* pages: 40–53, 1995.

[97] Steve W. Bova, Clay P. Breshears, Henry Gabb, Bob Kuhn, Bill Magro, Rudolf Eigenmann, Greg Gaertner, Stefano Salvini and Howard Scott. Parallel Programming with Message Passing and Directives. *Computing in Science and Engineering,* 3(5): pages = 22–37, 2001.

[98] R. Clint Whaley and Jack J. Dongarra. Automatically Tuned Linear Algebra Software. *Technical Report UT-CS-97-366,* University of Tennessee, 1997.

[99] Isom L. Crawford and Kevin R. Wadleigh. Software Optimization for High Performance: Creating Faster Applications. *Prentice Hall PTR,* ISBN:0130170089, 2000.

[100] Kevin Dowd and Charles Severance. High Performance Computing, 2nd Edition. *O'Reilly & Associates,* ISBN:156592312X, 1998.

[101] Anthony LaMarca and Richard E. Ladner. The Influence of Caches on the Performance of Sorting. *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms,* pages: 370–379, 1997.

[102] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala and Mithuna Thottethodi. Recursive Array Layouts and Fast Parallel Matrix Multiplication. *Proceedings of the 11th ACM Symposium on Parallel Algorithms and Architectures,* pages: 222–231, 1999.

[103] Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra and Mithuna Thottethodi. Nonlinear Array Layouts for Hierarchical Memory Systems. *Proceedings of 1999 ACM International Conference on Supercomputing,* pages: 444–453, 1999.

[104] Jeremy D. Frens and Davis S. Wise. Auto-Blocking Matrix-Multiplication or Tracking BLAS3 Performance from Source Code. *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* pages: 206–216, 1997.

[105] Hans Sagan. Space-Filling Curves, *Springer-Verlag,* ISBN:0387942653 ,1994.

[106] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik,* 13, pages: 354–356, 1969.

[107] Mithuna Thottethodi, Siddhartha Chatterjee and Alvin R. Lebeck. Tuning Strassen's Matrix Multiplication for Memory Efficiency. *Proceedings of SC98, High Performance Networking and Computing,* 1998.

[108] Matteo Frigo, Charles E. Leiserson, Harald Prokop and Sridhar Ramachandran. Cache Oblivious Algorithms. *Proceedings of the 40th Annual Symposium on Foundation of Computer Science(FOCS'99),* pages: 285–298, 1999.

[109] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson and Keith H. Randall. Dag-consistent Distributed Shared Memory. *Proceedings of the Eighth ACM Symposium on Parallel Algorithms and Architectures(SPAA),* pages: 297–308, 1996.

[110] Vipin Kumar, Ananth Grama, Anshul Gupta and George Karypis. Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms. *Addison-Wesley Pub Co.,* ISBN:0805331700, 1994,

[111] Hyuk-Jae Lee, James P. Robertson and J.A.B. Fortes. Generalized Cannon's Algorithm for Parallel Matrix Multiplication. *Proceedings of the 11th ACM International Conference on Supercomputing,* pages: 44–51, 1997.

[112] Meng-Shiou Wu, Ricky A. Kendall and Srinivas Aluru. Mixed Mode Matrix Multiplication. *IEEE International Conference on Cluster Computing (CLUSTER'02),* pages: 195–204, 2002.

[113] Meng-Shiou Wu, Ricky A. Kendall and Srinivas Aluru. A Tunable Collective Communication Framework on Cluster of SMPs *Proceeding of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN),* pages: 56–63, 2004.

[114] Meng-Shiou Wu, Ricky A. Kendall and Srinivas Aluru. Exploring Collective Communications on a Cluster of SMPs. *Proceeding of 7th International Conference on High Performance Computing and Grid in Asia Pacific Region (HPCAsia),* pages: 114–117, 2004.

[115] Meng-Shiou Wu, Ricky A. Kendall and Kyle Wright. Optimizing Collective Communications on SMP Clusters. *the Proceedings of the 2005 International Conference on Parallel Processing (ICPP-05),* pages: 399-407, 2005.

[116] Meng-Shiou Wu, Ricky A. Kendall, Zhao Zhang and Kyle Wright. Performance Modeling and Tuning Strategies of Mixed Mode Collective Communications. *Proceedings of the ACM/IEEE Supercomputing Conference (SC05),* 2005.

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this dissertation. First and foremost, Dr. Ricky A. Kendall for his guidance, patience, and support throughout this research and the writing of this dissertation. Next, thanks are in order to Dr. Zhao Zhang for his guidance and support as my co-advisor and committee member. I would also like to thank my other committee members for their efforts and contributions to my graduate education: Dr. Mark Gordon, Dr. Brett Bode, and Dr. Diane Rover. I would like to thank Dr. Srinivas Aluru for his unique insights in our work together.

I would like to thank Scalable Computing Laboratory at Ames Laboratory for providing me with computing resources during my research. I would like to thank the Department of Electrical and Computing Engineering office staff for helping me with all the administrative procedures.

Lastly, I thank my family for their support with a special thanks to my wife Mei-Ling, who has supported me throughout my graduate education and because of her unselfishness, has allowed me to pursue my academic and personal goals vigorously.