5-12-2001

Overlapping of Communication and Computation and Early Binding: Fundamental Mechanisms for Improving Parallel Performance on Clusters of Workstations

Rossen Petkov Dimitrov

OVERLAPPING OF COMMUNICATION AND COMPUTATION AND EARLY BINDING: FUNDAMENTAL MECHANISMS FOR IMPROVING PARALLEL PERFORMANCE ON CLUSTERS OF WORKSTATIONS

By

Rossen Petkov Dimitrov

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Science
in the Department of Computer Science

Mississippi State, Mississippi

May 2001

OVERLAPPING OF COMMUNICATION AND COMPUTATION AND EARLY BINDING: FUNDAMENTAL MECHANISMS FOR IMPROVING PARALLEL PERFORMANCE ON CLUSTERS OF WORKSTATIONS

By

Rossen Petkov Dimitrov

Approved:

Anthony Skjellum Associate Professor of Computer Science (Director of Dissertation)	Rainey Little Associate Professor of Computer Science (Committee Member)
Donna S. Reese Associate Professor of Computer Science (Committee Member)	Ioana Banicescu Associate Professor of Computer Science (Committee Member)
Arkady Kanevsky Adjunct Associate Professor of Computer Science (Committee Member)	A. Wayne Bennett Dean of the College of Engineering

Name: Rossen Petkov Dimitrov

Date of Degree: May 12, 2001

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Anthony Skjellum

Title of Study: OVERLAPPING OF COMMUNICATION AND COMPUTATION AND EARLY BINDING: FUNDAMENTAL MECHANISMS FOR IMPROVING PARALLEL PERFORMANCE ON CLUSTERS OF WORKSTATIONS

Pages in Study: 310

Candidate for Degree of Doctor of Computer Science

This study considers software techniques for improving performance on clusters of workstations and approaches for designing message-passing middleware that facilitate scalable, parallel processing. Early binding and overlapping of communication and computation are identified as fundamental approaches for improving parallel performance and scalability on clusters. Currently, cluster computers using the Message-Passing Interface for interprocess communication are the predominant choice for building high-performance computing facilities, which makes the findings of this work relevant to a wide audience from the areas of high-performance computing and parallel processing.

The performance-enhancing techniques studied in this work are presently underutilized in practice because of the lack of adequate support by existing message-passing libraries and are also rarely considered by parallel algorithm designers.

Furthermore, commonly accepted methods for performance analysis and evaluation of parallel systems omit these techniques and focus primarily on more obvious communication characteristics such as latency and bandwidth.

This study provides a theoretical framework for describing early binding and overlapping of communication and computation in models for parallel programming. This framework defines four new performance metrics that facilitate new approaches for performance analysis of parallel systems and algorithms. This dissertation provides experimental data that validate the correctness and accuracy of the performance analysis based on the new framework. The theoretical results of this performance analysis can be used by designers of parallel system and application software for assessing the quality of their implementations and for predicting the effective performance benefits of early binding and overlapping.

This work presents MPI/Pro, a new MPI implementation that is specifically optimized for clusters of workstations interconnected with high-speed networks. This MPI implementation emphasizes features such as persistent communication, asynchronous processing, low processor overhead, and independent message progress. These features are identified as critical for delivering maximum performance to applications. The experimental section of this dissertation demonstrates the capability of MPI/Pro to facilitate software techniques that result in significant application performance improvements. Specific demonstrations with Virtual Interface Architecture and TCP/IP over Ethernet are offered.

DEDICATION

I would like to dedicate this work to my wife Albena, my daughter Andrea, and to my parents.

ACKNOWLEDGEMENTS

The author expresses his special thanks to his major advisor, Dr. Anthony Skjellum, for his invaluable guidelines and support throughout the entire period of this research. Also, thanks are due to MPI Software Technology, Inc. for the facilities provided to the author used in the development and experimental portion of this work as well to Ms. Amy Ramsey who helped with copy-editing this dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Modern science and technology have created opportunities for achieving greater productivity and efficiency than ever before. To a large degree, this has become possible because of rapid advancements in the simulation and prediction of natural phenomena and complex physical processes that have occurred in the past several decades. Major facilitators of these advancements have been numerical analysis and high-performance computing. Numerical simulation techniques generate large data sets and require intensive computations for reaching acceptable levels of precision and speed of processing. Increasing computational capabilities is essential for continued progress of modern science and technology in all fields.

Parallel processing has been recognized as one of the fundamental approaches to achieving high-performance computing. In past years, the continuous growth of computing power of standalone systems has not diminished the importance of parallel architectures. On the contrary, with increasing accessibility of commercially available computer and network components, a new generation of parallel systems has attracted the attention of a broader user base. In the past, only a limited number of government laboratories and large businesses had access to high-performance parallel systems,

primarily because of the tremendous cost of these systems, both to purchase and to operate. Nowadays, with the rapid growth of microprocessor power, reduction of memory prices, emergence of high-speed networks, and availability of parallel system and application software, building and maintaining parallel systems have become cost effective and thus, accessible to a considerably wider range of users. The introduction of distributed parallel systems based on groups of networked workstations and personal computers, commonly called clusters of workstations, has amplified this recent trend. Currently, cluster computing represents one of the fastest growing segments of the high-performance computing industry. This study focuses on defining, justifying, and demonstrating software mechanisms for achieving high-performance, scalable parallel computing on clusters of workstations interconnected with high-speed networks.

In this chapter, background and statement of the problem are presented. Then, the objectives and justification of the study are outlined. Finally, the scope of the study and plan of presentation are described.

1.1 Background

Improving the performance of parallel systems has been a major goal of hardware and software designers since the inception of parallel processing. Although the idea of using multiple processors for speeding up computations has been adopted since the early ages of computers, achieving high-performance, scalable, and efficient computing has proven to be an enduringly challenging task. In addition, modeling and performance analysis of parallel systems are significantly more complex than in sequential systems.

This section reviews fundamental approaches for improving performance and introduces key concepts of performance analysis in parallel systems. Furthermore, the importance of parallel models and performance metrics is identified.

Throughout this study, the term "parallel system" is used to represent the combination of four components:

- Processing elements with memory (processors),
- Network interconnect that provides physical links between processors,
- Low level system software, such as firmware and device drivers, for performing basic data transmission services, and
- A middle software layer that provides communication abstractions and an interface to applications.

Application-level software and the performance capabilities of the processing elements are not among the targets of this study. Rather, attention is focused on the other three components of the parallel system, with special emphasis on the performance and scalability attributes of the communication middleware layer. An overview of general approaches for performance improvement is presented. This overview is used to illustrate the general area of focus of this work.

1.1.1 Approaches for Increasing Parallel Performance

The approaches for improving parallel performance can be broadly divided into two major groups: hardware and software approaches. The hardware approaches are in turn divided into approaches that target the performance of computers as standalone

components and approaches for improving network performance. Recent advancements in microelectronic technologies and processor architectures have led to a rapid increase in the computing capabilities of microprocessors. Modern, super-scalar, pipelined processor architectures with specialized vector units, such as the Intel Pentium III SSE and Motorola PowerPC G4 AltiVec, are among the leading technologies for increasing computer performance. New technologies in the area of memory and I/O subsystem architectures, such as Rambus and InfiniBand, are also being investigated and applied in practice for further improvement of the computing capabilities.

The networking aspect of hardware optimizations focuses on building scalable parallel architectures by employing high-speed communication links and efficient topologies. These capabilities have been among the most important technological achievements of supercomputers. Massively parallel processor (MPP) platforms, such as the Cray T3D/T3E, use multihundred-megabyte-per-second interconnects and multi-dimensional topologies in order to reduce the communication overhead associated with parallel processing (Andersen et al. 1997). The increase of raw link speed reduces communication time, thus improving parallel efficiency and overall application performance.

Common choices of MPP network topologies are multi-dimensional meshes and hypercubes. Providing multiple communication links per processor minimizes the probability of packet conflicts, and thus, reduces the communication overhead and further improves performance and scalability. An important hardware architecture that leads to the increase of both effective processing power and communication performance is the

two-level multicomputer (Boden et al. 1995). This architecture introduces a specialized communication processor that offloads the main processor from communication activities and also achieves higher sustained communication bandwidth. An example of two-level multicomputer system is the Intel Paragon, which is based on compute nodes with two i860 processors, one of which is used as a dedicated communication processor (Sprangers et al. 1995); other versions of this Paragon architecture have also been developed.

In the past several years, clustering workstations and personal computers has led to broadening the use of low-cost parallel systems. An important factor for achieving high computation efficiency in clusters is the performance of the network interconnects. Initially, clusters were primarily based on Ethernet and ATM networks using the TCP/IP transport. However, the communication capabilities of these networks were soon recognized as major performance bottlenecks for clusters. High-speed networks such as Myrinet (Boden et al. 1995) and Giganet cLAN (Giganet 1999) emerged to address the performance gap between computers and networks. These high-speed networks provide an order-of-magnitude increase in communication performance, which has served as a main facilitator of high-performance cluster computing. Some of the major architectural characteristics of these networks are the use of high-degree, non-blocking cut-through switches and I/O-bus-master-capable, intelligent network interface controllers (NIC) equipped with on-board direct-memory-access (DMA) engines. Cut-through switches introduce minimal hardware overhead during packet switching, thereby facilitating low latency and high bandwidth of communication links. The DMA engines allow the NIC to access the system memory without the participation of the main CPU and deliver data

directly into user buffers. As such, the NIC of high-speed networks serves as a communication processor that complements the main CPU, and consequently, allows a workstation to be viewed as a two-level multicomputer (Boden et al. 1995). The memory hierarchy of modern workstations, and especially its cache coherency, plays an essential role in this two-level multicomputer architecture. Cache coherency maintains the consistency of user data by invalidating the cache lines that contain memory locations modified by the NIC DMA engines.

The software approaches for increasing parallel performance can also be subdivided into two subgroups: application-software and system-software approaches. The application software approaches rely on employing parallel algorithms with minimal asymptotic complexity of computation as well as a minimum number and/or volume of communication transfers. Often, a balance between computation and communication performance is necessary in order to achieve optimal overall performance. Another major requirement of parallel application development is to preserve maximum portability of the code across numerous platforms. These needs have been addressed by parallel programming models, such as BSP (Valiant 1990) and LogP (Culler at al. 1996). These models help parallel application developers to assess the expected performance of their algorithms on different target platforms and evaluate the trade-offs between computation and communication performance as well as between performance and portability.

Another mechanism for improving applications' performance is tuning the algorithms to reflect the topology of the hardware platform. This tuning can result in a graph of the communication transactions that best matches the network topology. For

instance, neighbor processes in a virtual Cartesian topology can be mapped to computer nodes that are connected to the same network switch, which will result in reduced network contention and faster communication. However, parallel applications that reflect specifics of the underlying platforms sacrifice code portability. Parallel system software can help such applications by creating topology abstractions and portability layers that are specifically targeted to different platforms.

Load balancing is another application software mechanism that is frequently used to improve the effective performance of parallel algorithms. This mechanism seeks to distribute all of the work among processors more evenly so that idle processor cycles are avoided. Load balancing is a mechanism that can also be implemented in system software that is transparent to applications.

Systems software approaches are the second subgroup of software approaches for increasing parallel performance. Two of the major goals of parallel system software are to provide portable, high-level communication and data-layout abstractions to application software and to utilize the capabilities of the hardware resources in an efficient manner by achieving low processing overhead. In order to meet the often contradictory requirements for performance, scalability, abstraction, portability, and flexibility, parallel system software has evolved into two, often disjoint directions: low-level system software providing optimized point-to-point data movement and higher-level middleware achieving abstraction and portability as well as providing collective communication primitives. Examples of low-level software systems are Active Messages (von Eicken et al. 1992), U-net (Basu et al. 1995), and Portals (Brightwell and Shuler 1996). Typical

representatives of the middleware layer are PVM (Geist et al. 1994) and MPI (Message Passing Interface Forum 1994; Gropp, Lusk, and Skjellum 1999). Common system software approaches for improving parallel performance and scalability are as follows:

- reducing communication overhead,
- eliminating intermediate data copies,
- minimizing operating system intervention in communication,
- reducing main processor participation in communication activities, and
- implementing scalable algorithms for collective operations.

1.1.2 Parallel Models

A number of models for representing parallel processing and parallel architectures have been used in practice. The first group of models focuses on the methods for exchanging data between the processors of a parallel system. Shared memory and message passing are the two major models in this group. Distributed shared memory systems are a special case of the shared memory architecture. A typical representative of the shared memory architecture is the SGI Origin 2000. The IBM SP2 is a distributed shared-memory machine while Intel Paragon is representative of message-passing architectures. Clusters of workstations also follow the message-passing model for data communication between processors. Presently, multiprocessor workstations are becoming common for their low cost per processor. A cluster based on multiprocessor nodes may exploit both the shared memory model of communication within a node and the message-passing model between nodes.

The second group of models focuses on the relationship between instructions and data. The most widely used architectures, according to this group of models, are the single instruction multiple data (SIMD) and multiple instructions multiple data (MIMD) models. Computers that follow the SIMD model are sometimes also called vector processors, and also include large-scale bit-oriented processors. In the "golden ages" of parallel processing, vector processors were the predominant architecture. In recent years, the MIMD model has gained wider acceptance than the SIMD model. A special case of the MIMD model is the single program multiple data (SPMD) architecture, where the same parallel program is executed concurrently on multiple processors working on different portions of the data set. The SIMD and MIMD architectures follow the data-parallel model for parallelization. The alternative model is the data-flow model, in which parallel processing is implemented through pipelining.

The third important group of models concentrates on modeling parallel programming and is used for performance analysis. These models seek to quantify and formalize the characteristics of a parallel platform by using a set of primitives in order to describe the computational and communicational processes. The goal of these models is the derivation of an expression that can be used to parameterize the performance of a parallel algorithm. When this expression is applied to a specific parallel architecture, it is expected to provide an accurate performance estimate of the application subjected to modeling. Important representatives of these models are the PRAM model (Fortune and Wyllie 1978), the BSP model (Valiant 1990), the LogP model (Culler et al. 1996), and the LogGP model (Alexandrov et al. 1995).

1.1.3 Metrics for Parallel Performance

Metrics are an important factor for quantitative performance analysis. Performance analysis of parallel systems is significantly more complex than the analysis of sequential systems. Execution time is the predominant performance metric for sequential systems. A broader set of metrics is necessary for precise and meaningful analysis of parallel systems. Execution time is still one of the most widely used parallel performance metrics. However, execution time alone is insufficient to give insight regarding the complex interactions between the hardware and software components of a parallel system. For this reason, metrics such as efficiency, scalability, and cost are introduced in parallel performance analysis. A more detailed description of these metrics is presented in Chapter II.

Often, parallel performance models use point-to-point metrics, such as round-trip time latency and one-way bandwidth, for describing the communication overhead and efficiency associated with parallel processing on a given platform. Studying communication efficiency is accepted as one of the fundamental approaches to understanding and predicting performance and scalability. Therefore, estimations of the performance characteristics of a parallel system are frequently based on point-to-point latency and bandwidth measurements. Further, it is considerably easier to measure latency and bandwidth through simple message exchange experiments between two processes, rather than perform a complex investigation and analysis of the behavior of the entire parallel system when collective or multiple concurrent pair-wise communication transactions are executed. One of the objectives of this work is to demonstrate the limited

descriptive power of point-to-point metrics, and especially short-message latency, when used as the sole factor in performance analysis of parallel systems.

1.1.4 Trends in Parallel Processing

In recent years, clusters of workstations have become the dominant architecture of choice for building parallel systems. In the early years of cluster computing, clusters were viewed primarily as low-cost architectures for utilizing parallel processing. Today, clusters interconnected with high-speed networks demonstrate performance and scalability capabilities attributable before only to MPP supercomputers. The low cost and performance potential of clusters have shifted the attention of the organizations that rely on parallel computing from supercomputing platforms to clusters. This study responds to this trend by specifically targeting its theoretical and experimental focus on clusters of workstations.

1.2 Problem Statement

This work addresses a number of problems in the areas of performance analysis on clusters, design and implementation of message-passing middleware systems for modern cluster solutions, and design of efficient parallel application software. These problems can be divided into the following four categories:

- insufficient theoretical description of important performance sources,
- lack of MPI implementations specifically targeted for clusters,
- legacy MPI codes tightly coupled to specific MPI implementations, and
- limited validity of common methods for performance analysis.

1.2.1 Insufficient Theoretical Description of Performance Sources

Although clusters resemble traditional MPP platforms, there are a number of differences in their characteristics, some of which significantly affect performance and scalability. Recent trends show that parallel system and application software are often transferred mechanically from MPP systems to clusters. Although this porting approach is low cost and allows for a quick transition of important applications to the new environment, it does not reflect important differences between clusters and MPP. These differences have significant effects on performance.

New sources of performance improvement must be studied and applied to clusters. These sources should revisit strategies abandoned earlier because of past inhibiting factors. Some of these sources, such as overlapping and early binding have not been theoretically described and quantified to a sufficient degree. The theoretical description of these mechanisms requires the development of new models for parallel processing. Also, new performance metrics that capture the effects of these mechanisms are necessary for meaningful quantitative analysis.

1.2.2 Lack of MPI Optimizations for High-Speed Clusters

MPI has become the most widely used interface for writing parallel algorithms with the message-passing paradigm. Vendors of major MPP platforms, such as IBM, SGI, and HP, offer native, platform-specific implementations of MPI. These implementations employ hardware and operating system optimizations that are specific to each vendor's MPP platform. In the public domain, MPICH, which is an implementation

from Argonne National Laboratory and Mississippi State University (Gropp et al. 1996), has become the implementation with the largest user base. Initially, MPICH was implemented with the idea to serve as a reference proof-of-concept MPI implementation and also to help refine the MPI specification.

Vendor-specific MPI implementations, and especially MPICH, are currently being ported to clusters with minimal, if any, architectural changes. The changes are usually limited to only the lowest software layers that interact with the network transport interfaces. These ports do not reflect the new characteristics of clusters and high-speed network architectures and as a result they miss opportunities to provide optimizations specifically targeted to clusters. New MPI implementations are needed to bridge the differences between traditional MPP systems and clusters. Earlier, clusters were viewed only as a cheap alternative to multicomputers, and providing a quick port of MPI was considered to be an adequate objective. Performance was not necessarily the main goal of these early ports. Presently, however, clusters are used as a major parallel platform for providing teraflop performance and, in many instances, clusters replace MPP platforms. Consequently, MPI implementations that reflect the specific characteristics of clusters are needed. This work presents a design and implementation of an MPI implementation that specifically targets clusters interconnected with high-speed networks.

1.2.3 Legacy MPI Codes

Over the past several years, MPI has become a de-facto standard for writing portable parallel applications. A large number of parallel codes have been ported to MPI

and, frequently, MPI is the middleware API of choice when new algorithms are to be implemented. The development process of parallel applications typically targets a specific platform. Thus, the applications naturally reflect the performance characteristics, assumptions, and capabilities of the underlying platform and the MPI implementation installed on this platform. When these applications are ported to a different platform, whether using a port of the same MPI implementation on which they were initially created or a new implementation, they continue to use the same performance mechanisms that were chosen on the first platform. This porting procedure disregards the fact that these mechanisms may not perform as well on the new platform. In fact, different mechanisms may provide a far greater opportunity for performance improvement.

In this manner, many legacy MPI applications have become tightly connected to the MPI implementation on which they were first created. As a result, the approaches to performance optimizations and the assumptions made in these initial MPI's have had great impact on application programmers and the codes that they have written. Consequently, the limitations of specific MPI implementations have been directly propagated into applications. Although this does not affect the portability of applications, it clearly does affect their performance when executed on different platforms or MPI implementations.

MPICH has provided a great resource to the parallel processing community to recognize the abstraction power and performance potentials of MPI. However, MPICH did not contemplate all of these potentials nor did it emphasize efficient architectures that enable various performance enhancing techniques. This fact has had a significant impact

on a large number of legacy MPI applications written using MPICH or MPI implementations derived from MPICH. Even widely used parallel benchmarks, such as the NAS Parallel Benchmarks (Bailey et al. 1991), are written in a manner that reflects these limitations. The limitations of MPICH include following:

- sub-optimal implementation of collective operations,
- insufficient optimizations for persistent mode of communication,
- inability to support multi-threaded user programs,
- high-overhead derived datatypes engine,
- sub-optimal implementation of the persistent mode of communication,
- polling-based message progress engine, and
- polling method for message-completion notification.

These limitations lead to minimal or completely absent performance gains from such advanced software mechanisms as overlapping of communication and computation, early binding, and asynchronous processing. As a result, if an application attempts to use some of these mechanisms with MPICH, this application will exhibit minimal or even negative performance improvement, regardless of the capability of the underlying hardware platform and/or network infrastructure to provide sufficient support for such mechanisms.

Because overlapping, early binding, and asynchronous processing are among the important performance improvement factors of clusters not captured by most other MPI implementations, the design and implementation of the new MPI implementation presented in this work specifically targets these advanced software mechanisms. This work demonstrates that applications that use these software mechanisms on clusters

running the new MPI implementation achieve significant performance gain, while preserving application portability. Furthermore, algorithms that do not use the mechanisms in question experience no performance degradation; hence, the opportunity for improving performance of applications that employ early binding, overlapping, and asynchronous processing will not result in performance degradation of legacy applications. This is an important feature of any high-performance system and is referred to as "performance transparency." Effectively, performance transparent mechanisms provide opportunities for "free" performance improvement.

1.2.4 Limited Validity of Common Performance Analysis Methods

A variety of metrics are used for evaluating performance of parallel algorithms and platforms. Parallel performance metrics can be effectively divided into two groups – metrics that measure point-to-point performance and metrics that view the parallel system as a whole. The latter group is denoted "aggregate metrics" in this work. Aggregate metrics are based on application execution time and reflect the contribution of each processor. These metrics not only emphasize absolute performance and but also weigh scalability and efficiency, which makes them powerful tools for studying the complex interactions in a parallel system.

Point-to-point metrics emphasize only the communication attributes of the parallel system, specifically, latency and bandwidth of interconnection links. Traditionally, point-to-point measurements are performed by ping-pong tests between two processes. With a little variation, this type of test has also been accepted in the area

of cluster computing. Because the asymptotic peak bandwidth of modern high-speed networks is usually similar and often bounded by the I/O bus throughput, the performance characteristics of the entire cluster are extrapolated solely from the ping-pong numbers for the message latency; that is, the lower the latency is, the higher the performance of the cluster is presumed to be. This approach has limited validity for several reasons:

- Latency affects only the exchange of short messages. Typical parallel applications that are based on the message-passing paradigm use medium to coarse-grain data parallel algorithms and the messages that they generate are in the range of tens of kilobytes to megabytes in length. Latency has minimal impact on such messages. The communication performance is determined mainly by bandwidth.
- It is frequently assumed that the performance parameters of the links, as measured by point-to-point metrics, are preserved across the parallel system during the execution of applications. This assumption is optimistic for a large number of practical systems because it ignores important scalability factors such as network contention, bisection bandwidth limitations, and communication and application software architectures.
- Ping-pong tests do not offer any insight about the costs paid by the system and applications software for achieving the lowest point-to-point latency. Message-passing middleware, such as MPI, inevitably makes architectural compromises in order to minimize latency. These compromises are not revealed by the ping-pong test, so exaggerating the impact of point-to-point latency disregards such sources

of performance as overlapping of communication and computation, independent message progress, optimized collective algorithms, and low CPU overhead. These mechanisms are usually sacrificed by MPI implementations that aim solely at the lowest ping-pong latency.

Point-to-point metrics, and specifically latency, are easy to understand and measure, but they do not offer enough insight about parallel performance, scalability, and efficiency of the target systems. More elaborate metrics and benchmarks are necessary for detailed analysis and comparison. This work addresses this need by providing a set of new metrics and methods for obtaining the experimental values of these new metrics.

Although the concepts of overlapping of communication and computation and temporal locality are quite common and have been intuitively used for improving parallel performance, few in-depth studies from a theoretical, systematic, and practical point of view have been presented. This has resulted in insufficient design support for these mechanisms in deployed parallel systems. The emphasis has been primarily on more direct performance gains, such as reducing point-to-point latency and increasing bandwidth. Often, this has been at the expense of more elaborate architectures that enable a high degree of overlapping and early binding.

More commonly than not, the drive for lowest latency has prevented system designers from implementing hardware and software architectures that enable the mechanisms studied here. This has naturally led to limited usage of overlapping and early binding in application software design. Application designers have not seen a clear benefit for optimizing their algorithms with these more sophisticated mechanisms;

consequently, they have sacrificed the potential performance gain for simplicity of the algorithmic implementations.

1.3 Thesis

Overlapping of communication and computation, low processor overhead, asynchronous processing, and temporal locality are major sources of performance improvement, on clusters of workstations interconnected with high-speed networks. The overall parallel application performance gain that is afforded by use of these mechanisms significantly outweighs any negative impact they may have on-point-to-point latency for short messages. Overlapping of communication and computation and early binding can be described in theoretical models used for performance evaluation and prediction. These models can offer a quantitative analysis of overlapping and early binding.

Point-to-point performance evaluation schemes that primarily emphasize lowest latency are insufficient for describing the complex interactions between hardware and software in parallel systems. Together with widely used aggregate metrics, such as parallel speedup and efficiency, these interactions can be exposed by new metrics for performance evaluation. This work proposes degree of overlapping, processor overhead, degree of asynchrony, and degree of persistence as formal metrics for evaluating parallel performance. These metrics reveal critical characteristics of parallel systems needed in order to support a wide range of performance enhancing software mechanisms. They can be used to predict the performance behavior of parallel algorithms using overlapping and early binding on clusters with a high degree of precision.

In order to deliver the performance benefits of overlapping, temporal locality, and asynchrony to application software, the entire communication stack of a parallel system, including the message-passing middleware, should be designed specifically in order to support these capabilities. Message-passing systems with insufficient provisions for overlapping and early binding limit the potential for performance gain of applications that use these mechanisms; hence, application programmers are discouraged from designing algorithms that utilize more sophisticated approaches to performance optimizations. On the contrary, message-passing middleware that accounts for overlapping and early binding can deliver a substantial portion of these performance-enhancing mechanisms to the application software.

In summary, this work will show the following:

- Early binding and overlapping are important performance enhancing mechanisms on clusters.
- Early binding and overlapping can be described in theoretical models for parallel computing.
- New metrics are necessary to describe the complex interactions between software and hardware in parallel systems.
- The message-passing middleware has a critical role for propagating overlapping and early binding capabilities of lower layers of the communication stack to the application layer.
- The MPI implementation presented here facilitates overlapping and early binding.

1.4 Objectives

The main objective of this work is to address the problems identified and demonstrate the validity of the hypotheses specified in the thesis. These objectives can be summarized as follows:

- Define the parallel environment that is the subject of the study. Although the concepts presented here are applicable to a variety of parallel systems, this work concentrates on clusters of workstations interconnected with high-speed networks.
- Identify specific characteristics of clusters and point out the differences between clusters and traditional MPP systems.
- Study the impact of point-to-point latency on applications performance and its descriptive power in performance analysis of clusters. Also, investigate the hypothesis that a cluster with the lowest point-to-point latency measured with a ping-pong test offers the highest performance and degree of scalability.
- Identify important mechanisms for improving performance of communications middleware and parallel applications on clusters.
- Develop a theoretical description of these mechanisms and present a model for formalizing and quantifying their impact on application performance.
- Study the factors that impact the efficiency of these performance mechanisms.
- Define new performance metrics, present formal expressions for these metrics, validate their descriptive power, and demonstrate experimental methods for obtaining their values.

- Demonstrate the importance of message-passing middleware for building high-performance scalable parallel systems. Communication software is one of the important components of clusters. It can offer a variety of performance mechanisms that propagate the capabilities of the hardware to user processes. If designed improperly, message-passing middleware may become a performance bottleneck and limit systems scalability.
- Design and develop an MPI implementation that provides specific optimizations for clusters interconnected with high-speed networks. This implementation will be used for proving the concepts developed in this work.
- Re-implement established algorithms to take advantage of this MPI implementation and the performance mechanisms identified in this work. Then, compare these new algorithms with the original codes and provide analysis of the impact on performance and scalability. These results will be used to confirm the hypothesis of this work.
- Develop guidelines for writing optimal parallel algorithms designed to work efficiently on a variety of parallel platforms, and specifically on clusters, by using overlapping of communication and computation, temporal locality, and asynchronous processing. Using these performance-enhancing mechanisms does not make the applications less portable or inadequate to traditional MPP systems nor does it lead to performance degradation on systems that do not provide architectural optimizations that support these mechanisms.

1.5 Justification of Study

Computing is becoming a factor of vital importance to all aspects of the modern economy. Government organizations, research laboratories, and businesses of various sizes are making large investments to increase their processing capabilities. Computing resources are viewed as an important factor to achieving competitive advantage in today's economy. Parallel processing has been accepted as one of the major approaches for addressing the global need for increased computing capabilities. For the past few decades, MPP systems such as IBM SP, Intel Paragon, SGI Origin, and Cray T3D/T3E have been mainly used for high-performance computing. However, historically, the high cost of these systems allowed only a small number of organizations, primarily national laboratories and large businesses, to access these resources. The low cost of clusters has created opportunities for a much wider user base. Presently, clusters of workstations have become the most popular architecture for building new parallel computing systems and a large number of legacy applications in areas such as biotechnology, seismic exploration, weather prediction, protein folding, crack propagation, chemical molecule matching, and CFD simulations have been ported from MPP systems to clusters. Cluster computing has demonstrated its economic effect and, consequently, studies in this area will potentially benefit a large number of users.

This work describes the specific characteristics of clusters and their differences from traditional MPP systems, outlines important requirements for efficient MPI implementations on clusters, and demonstrates paths for increasing performance and scalability of parallel applications. The results of this work may constitute a significant

contribution to parallel processing theory and practice on clusters in the areas of performance evaluation, design and implementation of MPI middleware, and writing efficient parallel algorithms. Also, the achievements of this work can be extended to parallel processing in general, which further widens its expected significance.

1.6 Scope

Although the analyses and conclusions of this study can be related to performance and scalability on most parallel systems, attention is concentrated on a specific parallel environment, processing paradigm, communication model, and mechanisms for improving performance. This section describes the scope of these areas for this study.

1.6.1 Parallel Environment

The parallel environment studied here is a cluster of workstations interconnected with high-speed networks, such as Giganet and Myrinet. These networks feature gigabit-per-second data-link rates and non-blocking, cut-through switches. The interface controllers of these networks are intelligent devices with bus-master capabilities on the host peripheral bus.

The cluster nodes used for the experiments presented in this study are IBM PC computers with Intel architecture processors running Linux RedHat or Windows NT/2000 operating systems. At present, the combination of these operating systems and Intel processors is the main choice for building clusters. Although the number of available processor architectures and operating systems is significantly larger, this study considers the ones selected here as representative of state-of-the-art technology.

1.6.2 Parallel Processing Model and Communication API

The communication paradigm for exchange of data between processors used in this work is message passing. Shared-memory communication is not studied here. The parallel processing model is single program multiple data (SPMD), a variation of the multiple instructions multiple data (MIMD) model. The API used for inter-process communication is MPI. The data-parallel paradigm is used for building the parallel algorithms studied in this work.

1.6.3 Performance Mechanisms

This work concentrates on specific mechanisms for improving performance: overlapping of communication and computation, temporal locality, and asynchronous processing. Other performance mechanisms, such as topology awareness, multi-device architectures, optimal collective algorithms, and scatter/gather capabilities for non-contiguous message transmissions are considered outside the scope of this study. Many of these would be suitable for related future investigation.

1.6.4 Parallel Performance Metrics

Metrics are a fundamental tool for performance analysis of parallel systems. This study considers established common performance metrics, such as execution time, speedup, parallel efficiency, and CPU overhead, as well as the newly introduced metrics, such as degree of overlapping, degree of persistency, segmentation efficiency, and degree of asynchrony. The definition of these metrics is provided in Chapter II and Chapter III, respectively. Cost efficiency is frequently considered as an important factor when an

organization acquires hardware and software for building a cluster. Cost and cost efficiency are outside the scope of this study. It is assumed that clusters are generally much less expensive than large MPP systems and this, together with their performance potentials, makes them the current parallel platform of choice.

1.7 Audience

Two major groups of cluster users can be identified. The users of the first group have worked on MPP systems in the past and now are upgrading their computing capabilities by building new clusters. The low cost of clusters has attracted a new group of organizations and individuals to parallel processing. Among these new users are small and mid-size businesses, small research groups, and resource-constrained independent organizations.

The two groups of cluster users often have different goals. To help distinguish these two groups and define the scope of this study, an evaluation function $F_d(P, C) = P^q/C^r$ is used, where P denotes general performance and C denotes cost. The parameters q and r represent the relative weights of performance and cost in the evaluation function. When selecting a cluster-based architecture, organizations with emphasis on performance use $q > r$, while organizations concerned primarily with price use $r > q$. This expression is a generalization of the widely accepted price-performance metric. By manipulating the values of q and r, the above-proposed expression allows for customized evaluation of parallel systems according to the specific needs of users. The audience of this study is identified as the organizations that give priority to performance (*i.e.*, $q > r$). The

professionals that may be interested in the results of this study are parallel application programmers, designers of low-level software communication layers, message-passing middleware developers, parallel performance theoreticians, and system administrators.

1.8 Plan of Presentation

First, a review of literature sources discussing topics related to this study is presented. Then, a theoretical framework for quantifying the major sources of performance studied here is described. This framework develops a formal representation of overlapping of communication and computation and temporal locality. Although these mechanisms are familiar to the parallel processing community, no attempt for theoretical formalization and quantification of their impact on performance has been made so far. As part of the framework, a set of new metrics is defined. These metrics play a central role in understanding the importance of the concepts discussed in this study.

Next, a design of a new MPI implementation with optimizations for high-speed networks is presented. This implementation features an architecture that allows for low CPU overhead communication and a high degree of overlapping. Specific optimizations for temporal locality and asynchronous processing are provided. In order to demonstrate the advantages of these mechanisms, the MPI implementation offers an alternative mode of operation, which represents traditional approaches for implementing MPI, specifically using polling for synchronization and message progress with high CPU overhead. By providing these two modes, this implementation creates an opportunity for detailed analysis and fair comparison of the performance mechanisms studied in this work. To the

best of the author's knowledge, the MPI implementation presented in this work is the first one that offers these two alternative modes of operation to users.

Further, a study of the impact of the selected mechanisms on parallel applications performance is presented. This study uses various techniques for evaluation and analysis. Some of these techniques are based on traditional tools and benchmarks, such as point-to-point latency and bandwidth measurements as well as popular parallel kernel and applications suites such as the NAS Parallel Benchmarks (Bailey et al. 1991). Selected well-known parallel algorithms are presented and modified to take advantage of the new MPI implementation and performance mechanisms to reveal their effects. New evaluation procedures are developed from the theoretical framework. These new techniques focus on the capability of a parallel system to provide a high degree of overlapping of communication and computation, low processor overhead, temporal locality utilization, and asynchronous processing.

Finally, this work presents guidelines for writing parallel algorithms using the mechanisms for improving applications performance studied here. These guidelines offer techniques that lead to performance enhancement while preserving portability of the algorithms and avoiding performance degradation on systems that do not provide these performance mechanisms.

CHAPTER II

LITERATURE REVIEW

This chapter presents a review of literature sources that consider issues related to the objectives of this study beginning with work in the area of parallel performance and scalability analysis. The emphasis here is focused on the definition of performance criteria, metrics for quantifying performance, and methods for obtaining the values of these metrics experimentally. Next, important models for parallel computation are discussed and evaluated with respect to the goal of this work. Among these models are the BSP and the LogP models. Furthermore, the architectures of some of the most widely used platforms for intensive parallel computing are presented and the characteristics of these platforms are compared with the characteristics of clusters. Next, literature sources that focus on modern high-speed networks are reviewed. Attention here is concentrated on network physical link parameters, topology, interface controllers, software communication stacks, and methods for performing and completing transmission requests. Thereafter, important software mechanisms for improving parallel performance are reviewed. Two groups of software mechanisms are identified: system level and application level mechanisms. Special attention is paid to the system level mechanisms. Next, review of the Message-Passing Interface and some of its popular implementations

are discussed. MPI has become the predominant communications interface for portable parallel programming on clusters and studying MPI shows important dependencies between the behavior of the parallel applications, the message-passing middleware, and the communication infrastructure. Finally, this chapter presents a review on research related to improving performance on clusters. Important achievements are identified as well as areas that have not been studied in sufficient depth and offer opportunities for further research.

2.1 Parallel Performance Analysis

Parallel computing has emerged as a method for solving large problems in an acceptable amount of time using available processing technologies. The major resources of a parallel system are the number of its processing elements and the memory associated with each element. It is expected that adding more processors to the configuration of a parallel system will lead to an increase in the overall system's performance. Ideally, the speed of processing would be proportional to the number of processors. However, the behavior of most practical systems deviates from this ideal, linear scaling. The reasons for this behavior include communication and synchronization overheads associated with the exchange of data between processors, load imbalances, and the presence of serial parts in parallel algorithms (Amdahl 1967).

Migrating from serial to parallel processing significantly increases the complexity of performance analysis. New measures for adequate representation and quantification of performance are necessary. As opposed to evaluating performance on serial machines,

these measures should reflect factors such as the number of processors, workload, characteristics of the communication infrastructure, synchronization, algorithm complexity, and impact of software stacks.

Scalability is among the most widely used measures for performance analysis of parallel systems. Scalability can be used to predict whether an existing parallel system can be extended with more processors while preserving its performance characteristics, or whether a parallel algorithm can be implemented efficiently on a larger system. For a given problem size, scalability analysis can determine the maximum achievable increase of performance as well as the optimal number of processors in the system. In order to reveal the complex processes in a parallel system, elaborate procedures for performance and scalability analysis are necessary. The extensive research in the area of parallel processing shows that it is difficult to analyze and compare performance and scalability across different systems. In addition, users have different perspectives on the characteristics that they wish to analyze as well as the criteria they use for assessment. As a result, a variety of performance and scalability metrics have been proposed and used in the theory and practice of parallel processing. Below, some of the widely accepted definitions and metrics are presented.

- System Size (p): The number of processor elements in a parallel system.
- Problem Size (W): The number of basic computational operations required for solving a problem. Also referred to as work.
- Serial Time (T_s): The execution time of the best-known serial algorithm for solving given problem.

- Parallel Time (T_p): The time elapsed from the start of the parallel computation to the moment when the last processor finishes (Grama, Gupta, and Kumar 1993).
- Serial Fraction (s): The ratio of the inherently serial part of an algorithm to its execution time on one processor.
- Degree of Concurrency ($C(W)$): The maximum number of computation tasks that can be executed simultaneously in a parallel algorithm working on workload W (Grama, Gupta, and Kumar 1993).
- Cost (pT_p): The product of parallel time T_p and system size p.
- Speed (V): The amount of work performed for a given unit of time, $V = W/T_p$ (Sun and Rover 1994).
- Speedup (S): The ratio between the serial time T_s and the parallel time T_p, $S = T_s/T_p$.
- Efficiency (E): The ratio of the speedup S to the number of processors p used in the execution, $E = S/p$.
- Parallel Overhead (T_{ov}): The time that all processors spent on overhead operations, $T_{ov} = pT_p - T_s$. It is assumed that T_s is spent entirely on useful computation.

The definitions and performance metrics described above will be used in the theoretical derivations and performance evaluations throughout this work. A set of new metrics will be introduced and defined in Chapter III. These new metrics capture software

and hardware interactions that play a critical role in the precise description of early binding and overlapping of communication and computation.

Among the first attempts for describing the behavior of parallel systems is the fixed-size speedup analysis presented by Amdahl (1967). He states that the upper bound of the speedup of a parallel algorithm with serial portion s is $1/s$, where $0 < s < 1$. Amdahl shows that increasing the number of processors in a system for a given problem size is not justifiable beyond a certain limit. However, this fixed-size speedup has been shown to be an insufficient performance measure for scalability (Gustafson 1988; Nussbaum and Agarwal 1991) and scaled speedup has been proposed instead. Gustafson (1988) is among the first to investigate the performance of a parallel system when the problem size is increased with the number of processors. He points out that for a certain class of applications it is appropriate to increase the problem size with the number of processors as long as the total execution time is constant. An example of such an application is the weather forecasting, where the size of the problem can grow arbitrarily by increasing the grid granularity, the frequency of the time steps, or the volume of air masses. Consequently, increasing the problem size in scalability analysis has been shown to be both an acceptable and beneficial technique, which is emphasized by the fact that multiprocessor platforms are used not only for running the same problems faster, but also for running larger problems in the same time (Nussbaum and Agarwal 1991).

Scaled speedup has served as a basis for one of the most commonly accepted definitions of scalability: the performance of a scalable parallel system is linearly proportional to the number of processors used in this system (Sun and Rover 1994).

Researchers have employed variations of this definition based on different performance metrics or on the distinction between parallel systems and algorithms. Using scaled speedup as a performance metric, Flatt and Kennedy (1989) investigate the impact of the overhead resulting from parallelism on the performance of multiprocessor systems. In addition to the upper bounds imposed by the serial portion of parallel algorithms according to Amdahl's law, Flatt and Kennedy (1989) show that for systems on which the communication and synchronization overhead grows as fast as $\Theta(p)$ the speedup is worse than linear. They furthermore show that for overhead growing faster than $\Theta(p)$, the speedup reaches a peak after which it becomes negative (*i.e.*, a slowdown is observed). Hence, for the latter types of overhead, the parallel execution time of an algorithm with a given problem size W has a minimum for a certain unique value $p = p_0$. This value is shown to be the solution of the equation:

$$p_0^2 \frac{dT_{ov}(p)}{dp} = T_p^{\min}, \tag{2.1}$$

where: T_{ov} is the overhead expressed as a function of the number of processors p and $T_p^{\min}$ is the minimum execution time.

Further, Flatt and Kennedy (1989) investigate the parallel cost of the system at the optimal point p_0. They show that at this point the system efficiency is relatively low - close to 0.5 (*i.e.*, the parallel system with the specified overhead function works in a non-cost optimal mode). They then suggest that the optimal point of operation be chosen so that the function $F(p) = E(p)S(p)$ is maximized, which is equivalent to minimizing the factor pT_p^2. For more precise analysis, Flatt and Kennedy investigate the product of the

weighted geometric mean of efficiency and speedup. Minimizing the function $F(p)$ yields an optimal value for the number of processors p_f. Flatt and Kennedy show that for $p < p_f$ the relative gain of speedup is higher than the corresponding relative increase of the cost while, for $p > p_f$, the relative increase of the cost is higher than the corresponding relative gain in speedup. They state that the "marginal value of each processor" is at most half a processor if more processors are added after the optimal point p_f.

Gupta and Kumar (1990) assert that the applicability of the analytical results presented by Flatt and Kennedy (1989) are unnecessarily limited for systems on which the overhead function grows faster than $\Theta(p)$. Gupta and Kumar generalize the scalability analysis for a broader class of overhead functions than Flatt and Kennedy. Gupta and Kumar state that for most practical applications the overhead function T_{ov} can be expressed as in the equation:

$$T_{ov}(p,W) = \sum_{i=1}^{n} c_i W^{y_i} (\log W)^{u_i} p^{x_i} (\log p)^{z_i}, \tag{2.2}$$

where: p is the number of processors, W is the problem size, c_i are constants, x_i and y_i are parameters greater than 0, and u_i and z_i are parameters with possible values of 0 and 1. In their scalability analysis, Gupta and Kumar investigate two cases in respect to the growth rate of the overhead function. In the first case, the overhead function grows at a rate $T_{ov} \leq \Theta(p)$. Then, the value p^{max} of p for which the execution time is minimum is determined by the degree of concurrency $p^{max} = C(W)$ and the maximum speedup is expressed by a formula of the form $S^{max} = f(W, T_o, C(W))$. In the second case, the rate of growth of the

overhead function is $T_o > \Theta(p)$. Here, the value p^{max} is found as $p^{max} = min(p_0, C(W))$ where p_0 is the solution of the differential equation:

$$\frac{dT_{ov}(p,W)}{dp} = T_p. \tag{2.3}$$

For $p = p_0$ the execution time T_p has its analytical minimum. However, Gupta and Kumar demonstrate that for certain algorithms, $C(W)$ may have a lower value than p_0 and, therefore, determine the value of p_{max}.

Gupta and Kumar further study a simplified version of the overhead function:

$$T_{ov}(p,W) = \sum_{i=1}^{n} c_i W^{y_i} p^{x_i}. \tag{2.4}$$

Using this simplified function, furthermore assuming that the summation over the index i is dominated by the jth term, they find analytical expressions for the optimal number of processors $p_0 = k * W^{\frac{1-y_i}{x_i}}$ and the optimal parallel efficiency $E_0 = 1 - \frac{1}{x_j}$. Here, E_0 is the system efficiency at the optimal point $p = p_0$ in which Tp has its minimum and $S = S_{max}$. With their analysis, Gupta and Kumar summarize the importance of the overhead function for the scalability analysis of parallel systems. They broaden the definition of overhead to incorporate the inherently serial portions of parallel algorithms, communication, synchronization, load imbalance, an algorithm's inherent concurrency, and contention for shared resources (Gupta and Kumar 1990; 1993).

As a scalability metric of parallel systems, Grama, Gupta, and Kumar (1993) propose the isoefficiency function. This function reflects the "capacity" of a system to

deliver linearly increasing speedup with increasing numbers of processors. This metric is also based on the scaled speedup and shows how a system can effectively accommodate the addition of more computing resources. Isoefficiency is a quantitative representation of the degree of scalability of a parallel system. The isoefficiency function is defined by the equation:

$$W = KT_o(p, W), \tag{2.5}$$

and shows how the problem size W should grow with increasing numbers of processors so that the efficiency E is kept at a certain constant level. It is asserted that a small value of the isoefficiency function implies that with increasing numbers of processors only small increments in W are sufficient to keep the efficiency constant. Such systems are called "highly scalable." On the other hand, a large isoefficiency function reveals a poorly scalable parallel system. Systems that do not have isoefficiency functions at all are non-scalable and for these systems the efficiency cannot be kept constant with increasing the number of processors.

In the performance analysis provided by Tang and Li (1990), it is shown that minimizing pT_p^2, and more generally pT_p^r, is equivalent to maximizing the ratio between efficiency E and the execution time T_p. In this analysis, the factor r determines the relative importance of the execution time and efficiency. A low value of r means that the efficiency is more important than the execution time and the optimal point of operation will use smaller number of processors having higher efficiency. Higher values of r shift the optimal point to a larger number of processors where the execution time is shorter, but efficiency is lower. Kumar and Gupta (1990) analytically find the optimal number of

processors for minimizing pT_p^r and also show that minimizing this metric "is asymptotically equivalent to operating at a fixed efficiency that depends only on the overhead function" and r. Thus, Kumar and Gupta show that the isoefficiency scalability analysis yields equivalent results to the analysis of optimizing the pT_p^r metric.

Using definitions of scalability similar to the ones presented earlier, Sun and Rover (1994) introduce a new scalability metric called *isospeed*. As opposed to previous work, however, this metric is not based on speedup – it is based on the measure of average unit processor speed obtained as the amount of work W performed in T_p time; Sun and Rover consider speedup as an inadequate metric for scalability analysis. The isospeed metric shows how the size of a problem should grow from W to W' with the increase of the number of processors from p to p' so that the average unit speed remains constant. The isospeed analysis yields $W' = W(p'/p)$ for algorithms with ideal scalability, equivalent to overhead $T_{ov}(p) = C$, and $W' > W(p'/p)$ in the general case with $T_{ov}(p) = f(p) \neq C$. This analysis reveals the interrelation between the problem size W, the system size p, and the speed of computation. This interrelation "provides the scalability information of the algorithm-machine combination" (Sun and Rover 1994).

Nussbaum and Agarwal (1991) take a different approach for defining and quantifying scalability. Their analysis is based on asymptotic speedup. They do not consider cost-effectiveness in their scalability analysis, although it is implicitly reflected in the flexibility of increasing the size of a parallel platform. In their analysis, Nussbaum and Agarwal introduce the idealized model of an exclusive-read, exclusive-write (EREW) parallel random-access machine (PRAM) (this model is reviewed in mode detail

in the next section). They make a clear distinction between the scalability of a parallel algorithm and the scalability of a parallel architecture and state that the algorithmic scalability can be determined through measuring the speedup on a machine with idealized communication structure such as PRAM. Their investigation is concentrated on the scalability analysis of parallel architectures. Nussbaum and Agarwal define scalability of parallel systems for a given algorithm as the ratio of the asymptotic speedup of this algorithm when run on a real system and the speedup of the algorithm when run on an EREW PRAM. Analytically, the speedup is expressed as:

$$\psi(W) = \frac{S_{real}}{S_{PRAM}} = \frac{T_{p^{PRAM}}}{T_{p^{real}}}. \tag{2.6}$$

A larger value of $\psi(W)$ means that the performance of the investigated parallel system shows performance closer to the performance of a PRAM and hence, this architecture is more scalable.

Luke, Banicescu, and Li (1998) propose *effectiveness* as a scalability metric that incorporates both cost and performance considerations, similarly to the isoefficiency function defined by Kumar and Gupta (1993). As opposed to previous efforts, the analysis of the optimal effectiveness is not based on the parallel overhead function, which is often a complex function of p, W, and a number of platform-dependent parameters. This makes the effectiveness metric relevant to practical applications for which deriving theoretical expressions for the overhead and isoefficiency may be an exceedingly complex task.

Li and Skjellum (1997) investigate a poly-algorithmic approach for solving parallel dense matrix multiplication on 2-dimensional virtual process grids. They show that for a range of practical environments, the asymptotic analysis yields locally sub-optimal solutions and that only an implementation based on a combination of algorithms can provide global optimality in the entire space of problem size and target platform configurations. Li and Skjellum provide a study on the crossover conditions and guidelines for efficient implementations of specific algorithms on a variety of target platforms.

2.2 Models for Parallel Computation

Designing efficient parallel algorithms requires detailed analysis of the communication and computation complexity of these algorithms, as well as the low-level hardware and software characteristics of the target platforms. Various models for parallel computation have been proposed to address these issues. Among the models initially proposed is the parallel random-access machine (PRAM) model (Fortune and Wyllie 1978). PRAM is a simple extension of the von Neumann model of sequential computers to parallel systems. PRAM concentrates on the inherent parallelism of the algorithms and abstracts the low-level hardware details. While PRAM offers the convenience of high-level abstractions to the designers of parallel algorithms, it has a limited descriptive power because it ignores important performance factors, such as communication overhead, network bandwidth, synchronization, contention, and interconnect topology. Later models have attempted to address these factors while preserving sufficient

abstraction. Specifically, this study reviews the bulk-synchronous parallel (BSP) model and the LogP model as instances of models that address the realistic description of the complex processes that determine the effective performance of a parallel system. These models have been recognized as the most successful models in the area of distributed parallel processing with message passing communication. The BSP model is proposed as a bridging model between algorithms and hardware on parallel platforms, by capturing common performance characteristics of a wide range of practical systems (Valiant 1990).

The computation in BSP is represented by a series of "supersteps" separated by a global synchronization phase. Each superstep is divided into a local computation phase and a communication phase. During the communication phase, the processors exchange all messages that belong to the superstep. A superstep completes when the local communication and computation have finished on all nodes. Supersteps cannot be overlapped; consequently, the total execution time is the sum of the execution times of all supersteps. The BSP model offers a simple characterization of the interconnection network by using two parameters: per-processors throughput (g) and synchronization latency (L). Using the BSP model, Bilardi et al. (1996) express the time of a superstep as the sum of three components: $T_{superstep} = w + gh + L$, where w is the maximum time spent on local computation and h is the maximum number of messages exchanged during the superstep by any processor.

Valiant (1990) identifies a number of algorithms that can be accurately described by the BSP model, specifically matrix multiplication, Fast Fourier Transform, and certain sorting algorithms. However, this model does not adequately capture asynchronous

processing and, for a number of algorithms, it enforces unnecessary synchronization. Asynchronous processing can be especially beneficial to algorithms with naturally asynchronous computation patterns, such as master-slave and unstructured data-parallel algorithms or to parallel systems on which the processors tend to work asynchronously even when the algorithms suggest synchronous processing. As shown later, clusters are an instance of parallel systems with such characteristics. Although not explicitly modeled, the BSP model allows for overlapping of communication and computation if the local communication and computation phases within a superstep can be pipelined.

The LogP model provides a more detailed view of the communication characteristics of distributed systems (Culler et al. 1996). This model focuses on presenting a precise, formal description of the communication performance of a wide range of parallel platform interconnects. LogP uses four parameters for representing communication: latency (L) – the time between the initiation of message transmission and the moment when the last byte of this message is deposited at the destination process memory, overhead (o) – the time that the central processor spends on message transmission or reception, gap (g) – the time between two successive sends, and number of processors (P). Culler et al. (1996) emphasize that the LogP model focuses on communication of messages with short sizes. LogP models the communication activities in a parallel system at a lower abstraction level than does the BSP model. As a result, LogP offers a more precise description of communication performance, and specifically, the impact of the communication overhead on scalability and overall application performance. Also, LogP eliminates the global synchronization step of BSP and provides

for asynchronous processing. LogP does not model computation; therefore, it cannot provide adequate representation of overlapping of communication and computation.

The LogGP model (Alexandrov et al. 1995) goes a step further than LogP in reflecting communication details by providing a description of long-message transfers by explicitly introducing the network bandwidth parameter *G*. This parameter represents the gap between bytes of the same message, while the *g* parameter is used similarly to the LogP model for describing the gap between short messages. While LogP specifically focuses on modeling parallel systems that primarily use short messages, LogGP recognizes that a large number of parallel applications rely on long data exchanges and that the communication time of long messages is predominantly affected by *G*. LogGP is applicable to a large class of medium to coarse grain data-parallel algorithms from the area of scientific computing. Ad hoc models equivalent to LogGP have existed for some time prior to LogGP's publication as well.

A number of models for parallel computation have been proposed in recent years. These models address trade-offs in a space defined by abstraction power (necessary for improved portability) and precise description of communication characteristics (necessary for accurate performance analysis). None of these models has offered explicit provisions for overlapping of communication and computation or temporal locality of communication transfers. Therefore, the scope of these models is considered insufficient for the purposes of this work. A new theoretical framework that captures these important performance sources is needed. Explicit accounting for overlapping and early binding in parallel computation models will achieve multiple goals:

- Establish more accurate performance prediction of parallel models,
- Facilitate deeper understanding of the complex interactions between the components of a parallel system,
- Enable objective evaluation of parallel systems' capabilities to support overlapping and early binding, and
- Facilitate efficient implementations of parallel algorithms that exploit these important sources of performance.

2.3 Parallel Architectures

This section first reviews some of the widely used large-scale multiprocessor and multicomputer architectures, often referred to as supercomputers. In the past decade, IBM SP2, Intel Paragon, TMC CM-5, Cray T3D/T3E, and SGI Origin 2000 have been accepted as the major MPP architectures. Specifically, Cray T3E and SGI Origin 2000 are reviewed. Next, the architectural characteristics of clusters used for high-performance computing are presented. Finally, this section identifies important differences between MPP systems and clusters.

2.3.1 Massively Parallel Processors

Cray T3E is a distributed shared-memory (DSM) multicomputer based on Alpha 21164 microprocessors (Anderson et al. 1997). Each node contains local memory and a network router. The topology of the interconnection fabric is a 3-D torus and can scale up to 2048 nodes. This topology provides direct connections with 6 neighbor nodes.

Communication links achieve a raw speed of 600 MB/sec in each direction. Cray T3E nodes have an advanced memory subsystem that allows for access throughput to local memory of 1.2 GB/sec. These nodes are equipped with logic that is specifically designed to optimize access to remote memory and minimize latency for synchronization with the other nodes. This logic extends the physical address space of the microprocessors to allow a large, directly addressable memory space. Also, the logic is used to improve effective data throughput by pipelining remote accesses and providing hardware scatter/gather capabilities. Remote memory accesses bypass the processor bus, which allows for simultaneous local and remote requests to the memory of a given node. The Cray T3E provides a *shmem* communication library that achieves 1 microsecond access time and 350 MB/sec effective bandwidth to remote memory. The high degree of communication links per node and the advanced architectural solutions of the memory subsystem allow Cray T3E to demonstrate high-performance and scalability for a wide range of parallel problems, including fine-grain data-parallel algorithms.

SGI Origin 2000 is a cache-coherent, non-uniform memory access (ccNUMA) multiprocessor, based on single- or dual-processor nodes with R10000 processors (Laudon and Lenoski 1998). The system can accommodate up to 512 nodes with 4GB of memory each. The distributed-shared memory (DSM) architecture of Origin attempts to meet both scalability and cost requirements. The cache coherence is maintained by a directory-based protocol, which is intended to eliminate scalability bottlenecks of earlier SGI platforms. Origin supports a globally addressable memory model with non-uniform access times. Special care is taken to reduce the variances of the local and remote

memory accesses. The topology of the interconnect fabric is a bristled, fat hypercube based on the Spider router chip. The routers have 6 bi-directional channels and employ wormhole routing with a specially designed mechanism for congestion control. Every pair of nodes is connected to one router (*i.e.*, every 4 processors share a router). Thus, a 32-processor configuration uses a single 3-D hypercube. This topology provides each processor with a direct link to 12 processors connected to 3 neighboring routers. In addition, the topology allows for direct connections between nodes on the opposite corners of the cube for further reduction of routing latency. The Origin 2000 achieves 310 nanoseconds access time to local memory, 540 nanoseconds access to remote memory when the remote node is connected to the same router, and 773 nanoseconds average access time to remote memory in a 32-processor configuration. The point-to-point bandwidth between two neighbor nodes is on the order of 350 MB/sec.

2.3.2 Clusters

The general architecture of clusters is based on a collection of common uni- or dual-processor personal computer class workstations interconnected with a switched network. Occasionally, the nodes of computing clusters are high-end servers with larger degree of internal parallelism. Currently, the predominant node count of clusters ranges from 16 to 32, but significantly larger clusters on the order of 256 to 512 nodes are being built. Projects for deploying clusters with more than one thousand nodes are underway also. Sandia National Laboratories is the leader in the area of ultra-scale clusters with its Cplant project, whose goal is to build a cluster with more than ten thousand uni-processor

nodes (Riesen et al. 1999). The Cplant project is performed under the DOE ASCI program.

The predominant processor architecture for cluster nodes is 32-bit Intel x86, now moving toward next generation 64-bit Itanium processors. The major reasons for this dominance of Intel architecture are the low cost of the cluster nodes and the abundance of software support, such as drivers, network support, compilers, debuggers, message-passing middleware, administrative tools, and performance monitoring tools. Machines based on the Alpha processor are also frequently used in clusters for their superior floating-point performance. Usually, clusters based on Alpha processors have a relatively small number of nodes because of their higher cost. The overall computing capabilities of Alpha clusters are compensated by the greater performance of the individual compute nodes. Most recently, Apple workstations based on the RISC PowerPC G4 processor architecture with an AltiVec vector unit are gaining momentum for their gigaflop-capable, single-precision floating-point performance and their low cost.

Linux is the most common operating system of choice used for building clusters for parallel processing. This operating system is based on UNIX and offers a great degree of flexibility for achieving the goals of cluster computing: low cost, maintainability, software support, and satisfactory performance. Also, Linux facilitates fast transition of system software and parallel applications from the UNIX-based operating systems of the MPP systems (for example, IRIX on SGI Origin 2000 and AIX on IBM SP2) to clusters. Microsoft Windows NT/2000 is also used in clusters (Microsoft Corporation 2001). The development environment and graphical support of this operating system have attracted a

number of organizations that prefer to use commercially supported software and that also find the Windows environment compatible with their goals. Among the leaders in the area of cluster computing with Windows is the Theory Center at Cornell University. This center has successfully operated its large-scale AC3 Velocity cluster since 1999 (Cornell Theory Center 2001). The author of this dissertation has developed the MPI implementation that is being used as message-passing middleware at AC3 (Dimitrov and Skjellum 2000). This MPI implementation is presented in detail in Chapter IV.

The interconnection network is among the most important factors that determine the performance and scalability of clusters. Initially, networks such as 10/100 Mbit/sec Ethernet and ATM and TCP/IP transport were used for building computational clusters. These initial clusters were primarily used for proof of concept and were not widely deployed in production environments. TCP/IP communication is performed within the scope of the host operating system and is based on kernel calls for data transfers and synchronization. The TCP/IP protocol stack has high processing overhead and introduces intermediate data copies in the internal operating system buffers. This stack is designed to provide high-level communication services and to facilitate software portability.

TCP/IP has evolved as an internet streaming protocol for communication over networks with high error rates. In addition to its major reliable and in-order delivery service, TCP/IP provides a number of traffic optimization mechanisms developed under the assumption of high packet loss. These assumptions are often in direct contrast with the capabilities of the network infrastructure used in high-performance computing

clusters, where local-area networks with close proximities are often used and the packet loss form signal noise and router contention is minimal.

The specific purpose and the performance attributes of TCP determine the relatively low communication performance of clusters using this protocol over traditional Ethernet and ATM networks. These networks in combination with TCP are still widely used for building low-cost clusters that demonstrate the concept of affordable parallel computing. Also, organizations whose applications exhibit highly concurrent structure with minimal communication requirements might find the performance of clusters based on TCP communication satisfactory. However, for the majority of the parallel applications that are executed on clusters, the performance of the communication subsystem plays a critical role for achieving high overall performance and scalability.

In the past several years, a number of high-speed networks have emerged as alternative cluster interconnects to Ethernet and ATM. The need for fast interconnects is primarily created by the needs of the communication intensive data-parallel applications that are ported from MPP to clusters. These new networks feature high data link rates and optimized software layers that facilitate low overhead and zero-copy data transmission. As a result, data exchange between cluster nodes is performed with an order of magnitude higher bandwidth and an order of magnitude lower latency than communication over traditional Ethernet using the TCP/IP software stack. The performance related architectural characteristics of high-speed networks and the communication system software for these networks are described in detail in a separate section of this document.

2.3.3 Architectural Differences Between MPP and Clusters

Although clusters can be viewed as simply another instance of platforms for high-performance parallel processing, there are significant differences between the architectures of MPP systems and clusters. One of the objectives of this study is to identify these differences and to justify the approach for performance evaluation of clusters. Below, some of the important characteristics of MPP systems and clusters are summarized and compared.

Characteristics of MPP systems:

- interconnection networks with multiple links per processor (often 4 or more),
- special hardware optimizations for efficient communication,
- specially designed memory subsystems,
- cross-system cache memory coherence hardware protocols,
- inter-processor communication uses direct channels between memory banks of remote processors,
- availability of redundant paths between pairs of processors,
- specially designed protocols for contention avoidance,
- communication paths do not traverse a low-speed peripheral I/O bus,
- highly predictable communication behavior,
- operating systems are specialized and optimized to support high-performance computing and provide optimized communication libraries,

- operating system's code running on compute nodes is often optimized and performs minimal interrupt handling from peripheral devices,
- user processes have direct access to low level communication primitives provided by the platform vendors,
- usually, only one process is scheduled per processor,
- Global communication and computation resources are uniformly distributed among nodes with equal processor speed and memory capacities.
- Algorithms with structured data distribution and temporal regularity tend to execute synchronously.

Main characteristics of clusters:

- One network interface controller per cluster node (possibly serving four or more processors),
- communication paths traverse a relatively slow peripheral I/O bus that often limits the maximum sustained bandwidth,
- limited memory bus throughput, especially on Intel 32-bit architecture platforms,
- efficient communication methods using bus-master-capable network controllers require pinning of use buffers in physical memory, which is a high-overhead operation involving the operating system kernel,
- for cost reasons, network topologies have lower and more irregular cross-section bandwidth characteristics than the interconnects of MPP systems – usually fat trees are used when cluster nodes exceed the number of ports on the switches,

- lack of support for redundant routes between nodes for contention avoidance,
- general-purpose operating systems on cluster nodes, such Windows or Linux, with sub-optimal process and thread context switch characteristics,
- high variance and low predictability of communication events,
- peripheral devices such as video, keyboard, mouse, and storage generate asynchronous interrupts at different rates,
- user processes have limited access to low-level communication primitives (high-speed networks with operating system bypass try to address this deficiency),
- computers in the same cluster often have different processor speeds and memory capacities,
- even regular and structured parallel algorithms tend to result in imbalanced execution.

It is obvious that a large number of the assumptions on MPP systems are not valid on clusters. This justifies a different approach to achieving high performance on clusters and also suggests different procedures and metrics for performance analysis.

2.4 High-Speed Networks

High-speed networks are essential components of computational clusters. Typical local area networks such as 10/100 Mbit/sec Ethernet and ATM are not capable of delivering enough bandwidth for achieving efficient communication required by intensive parallel computations. Among the first high-speed networks are SCI (Dolphin Interconnect 2001) and Myrinet (Boden et al. 1995). The initial physical data rates of

these networks exceeded 1 Gbit/sec. Later, Giganet cLAN (Giganet Incorporated 1999) and ServerNet (Compaq Computer Corporation 2001) emerged as alternative choices. Giganet and ServerNet are compliant with the Virtual Interface Architecture (VIA) industry standard (Compaq Computer Corporation, Intel Corporation, and Microsoft Corporation 1997). This standard specifies the hardware and software interfaces between a high-speed network and a host computer system. Myrinet and Giganet have gained the widest acceptance as interconnects in high-performance clusters for their high data rates, scalability properties, and cost competitiveness. This section reviews Myrinet and Giganet technologies, as a representative subset. The next section presents the software interfaces of these networks used by upper layers for interaction with the network hardware. These software interfaces play a major role in building protocol stacks that enable low overhead, efficient communication.

2.4.1 Myrinet

Myrinet is a high-performance, packet-switching local-area network with gigabit-per-second data rates. The Myrinet building components are host interface controllers and high port-count non-blocking switches, which employ cut-through switching and source routing. Cut-through switching eliminates packet buffering in switches and improves the effective link throughput. Source routing uses the leading word of the packet for selecting the outgoing switch port. This mechanism allows for rapid routing without searching through lookup tables. The hardware latency introduced by each switch is on the order of 0.5 microseconds (Boden et al. 1995). Myrinet switches provide conflict-free connections

at full bandwidth to all links, unless the outgoing flows are directed to the same port (Myricom Incorporated 1997). A Myrinet link is a full-duplex channel that operates at an aggregate data rate of 2.56 Gbit/sec. Myrinet supports large packets, which enables easy support of higher-level protocols, such as TCP/IP, without adaptation layers. Because of its high data rate, low latency, and scalability, Myrinet is well suited for connecting computers into clusters used for intensive parallel computations. At present, Myrinet is the most widely used network for building high-performance clusters.

Myrinet host adapters are fully operational, independent microprocessor systems based on the LANai chip (Boden et al. 1995). LANai executes a Myrinet Control Program (MCP) that performs data transmission over the network and notifies the host system for completion of communication events. The MCP can be loaded and controlled by processes running in user mode. The MCP offers unique flexibility for implementing various low overhead software architectures. This mechanism, enabled by memory mapping of a Myrinet controller's resources in user-space memory (Boden et al. 1995; Dimitrov, Skjellum, and Protopopov 1997), has allowed a number of research groups to build high-performance, low-level messaging layers used in clusters. These projects have offered a great variety of solutions for optimal message passing with low communication overhead. Some of these projects are reviewed later in this chapter.

The local memory of Myrinet adapters is accessible from the on-board local bus (LBUS) and the external peripheral bus (EBUS). On-board memory can be mapped in user space and thus made accessible directly to user processes. This mapping is performed in two steps: first the operating system maps the NIC address space into kernel

space, and then on a request by the user, the kernel region is mapped into user space. Thus, by simply accessing certain addresses, the user process can manipulate NIC local memory and the registers of the LANai processor. To increase data transfer throughput, Myrinet interface adapters are equipped with three DMA engines. Two of the DMA engines are associated with the physical network interface: one for receiving packets from the network and one for sending packets in duplex mode. The third DMA engine can operate simultaneously with the first two, and is used for bi-directional data transfers between the local adapter memory and main memory (Myricom Incorporated 1997).

The LANai processor can operate in host peripheral bus master mode. Using this mechanism, LANai can access user buffers across the peripheral bus without the participation of the host CPU. This enables concurrent execution of computation activities performed by the central processor and communication activities performed by the network processor. This architecture is referred to as “two-level multicomputer” (Boden et al. 1995). The two-level multicomputer architecture frees the host processor from immediate responsibilities for data transmission and creates opportunities for overlapping of communication and computation. This study is focused on high-speed networks with architectures similar to Myrinet because of its capability to perform communication independently of the host system, which allows the main CPU to compute while the network controller is transferring data. This capability is the main facilitator of low CPU overhead and a high degree of communication and computation overlapping.

2.4.2 Giganet cLAN

Giganet cLAN is a high-speed, packet switched network with intelligent bus-master capable interface controllers (Giganet 1999). The raw link data rate of Giganet is 1.25 Gigabit/sec. Giganet uses eight- or thirty-port switches, which introduce hardware latency on the order of 0.5 microseconds. The data-link layer operates with small size cells, similarly to ATM networks. This allows for fair and rapid routing of packets on the fabric. Giganet emphasizes low-latency, high throughput communication while using minimum cycles of the central processor for communication. The low CPU overhead of Giganet is one of its most appealing factors for high-performance parallel processing on clusters with overlapping of communication and computation.

As opposed to Myrinet, the communication processor of Giganet controllers is not programmable by the user. The firmware of the coprocessor is implemented with ASIC technology. This feature limits the flexibility of Giganet NIC. On the other hand, the Giganet NIC processes incoming and outgoing packets faster because, in contrast to Myrinet, the network control program is executed in hardware. Giganet was the first fully operational hardware implementation of the VIA specification. Giganet implements a high degree of protection and security while still providing operating system bypass on the critical data paths. Giganet also offers dynamic auto-configuration of the connection topology. This increases its reliability and availability. The VIA software interface of Giganet uses a connection-oriented communication model while Myrinet's interface is connectionless. The connection-oriented model of Giganet, and VIA in general, is viewed as a scalability-limiting factor on large-scale clusters (Brightwell and Maccabe 2000).

2.5 Communication Interfaces and Software Stacks

Communication software stacks provide application processes with low-level and high-level services. The low-level services include error detection and correction, reliable in-order delivery, and completion notification of communication events. The high-level services include buffer management, message multiplexing and demultiplexing, user-level flow control, collective operations, and topology abstractions. The architecture of communication software has a significant impact on message-passing bandwidth and latency as well as on processor overhead. An excessive number of intermediate software layers may substantially degrade the communication performance delivered to application processes. Traditional protocol stacks isolate user processes from the I/O devices by imposing kernel API for protection. This section reviews the fundamental differences between the traditional protocol stacks and the recently introduced, low-overhead messaging layers with operating system bypass. Specifically, U-net, SHRIMP, Fast Messages, and the Virtual Interface Architecture are reviewed.

2.5.1 Approaches to Communication Software Stacks

Traditional communication software approaches are often based on the TCP/IP protocol stack, which closely follows the seven-layer ISO OSI reference model. According to this model, the software stack is divided into clearly separated layers with specific interfaces and services that they provide to upper layers and expect from lower layers. This architecture offers a high-level of abstraction and increases the degree of portability of the protocol components. A substantial portion of traditional stacks is

implemented in the operating system kernel. This increases the level of protection in the system and also helps guarantee fairness in the access to I/O resources. However, this separation of highly modularized layers delegating functionality to the kernel incurs intermediate data copies and also imposes high software overhead caused by crossing multiple interfaces and by process context switches. Each data copy at the boundaries of the software layers degrades effective communication performance. Alternative communication software architectures attempt to bypass the operating system during the performance-critical activities. This functionality is achieved by employing thinner protocol stacks with collapsing software layers and providing direct access of user processes to network hardware resources. The main concepts of the traditional and low-overhead communication stacks are differentiated in Figure 2.1. The shaded areas represent software modules that are typically implemented in the kernel.

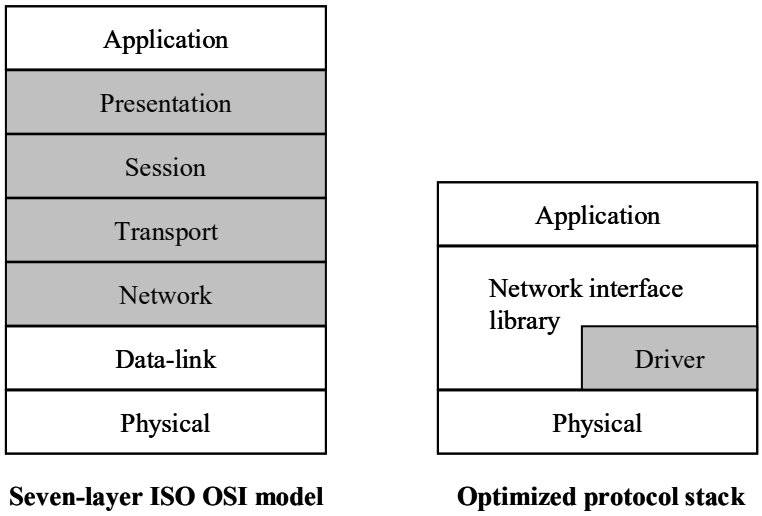


Figure 2.1 Comparison of traditional and optimized protocol stacks

2.5.2 U-net

U-net provides a model for building the communication software protocol stack in user space, thereby bypassing the operating system on the critical data paths (von Eicken et al. 1995). U-net's architecture presents a virtual view of the network interface to user processes. After an initialization phase, each process has direct access to the communication infrastructure of the network. U-net was originally developed for switched ATM networks with Fore Systems SBA 100 and SBA 200 interface controllers. These controllers use a programmable i960 processor, which serves as a network coprocessor (von Eicken et al. 1995).

The architecture of U-net demonstrates that a cluster of workstations interconnected with a fast network can achieve communication performance comparable to that of supercomputers. Specifically, U-net addresses the overhead and latency of short message transfers. U-net recognizes the trend for building intelligent network interfaces with local processing capabilities and local on-board memory. Tapping these resources requires a different approach to communication software. Instead of treating the NIC as a passive device controlled by the kernel, U-net relies on cooperation between the central processor executing the user process and the network coprocessor controlled by its own logic specifically designed for efficient communication.

The architecture of U-net is based on interface *endpoints*, which serve as user process handles to the network interface, and *communication segments* (Basu et al. 1995). The communication segments are regions of memory used for holding message queues. Send and receive message requests are created as descriptors in the communication

segments and then added to the message queues. U-net uses three queues – send, receive, and a queue for free descriptors. The send and receive queues serve as an interface between the user process and the NIC. The NIC removes descriptors from the queue and executes the operation requested by the user process. U-net is capable of achieving zero-copy transfers between user space buffers that belong to processes on different network nodes.

U-net/MM (Welsh, Basu, and von Eicken 1997) is an effort that extends U-net. U-net/MM eliminates U-net's restriction to place data buffers only in the communication segments. U-net/MM provides users with an interface for pinning pages of user buffers in physical memory on demand. Thus, a user can initiate a zero-copy transfer from any virtual address in user space, unlike U-net, which uses astatically allocated segments. This flexibility is achieved by a dynamic virtual-to-physical address translation mechanism. The translations are performed by the U-net/MM kernel module and are stored in a translation look-aside buffer (TLB) maintained in the NIC memory and synchronized with the virtual memory mechanism of the operating system. U-net/MM addresses issues related to TLB misses and TLB coherency by using a mechanism for performing dynamic translation of missing pages and invalidating translations released by user processes when communication to/from the buffer has completed. The TLB misses cause a substantial overhead, but their occurrence is relatively rare, so the increased flexibility of U-net/MM is viewed as an advantage over U-net. The dynamic translation mechanism first introduced in U-net/MM was later used as one of the foundations of the VIA specification.

2.5.3 SHRIMP

The Princeton SHRIMP multicomputer project demonstrates a software architecture that achieves low-latency and high-bandwidth communication through virtual memory-mapped network interfaces (Blumrich et al. 1994). This architecture specifically targets the system overhead associated with message initiation. Additionally, by employing a user-level block transfer mechanism, SHRIMP allows for high-sustainable bandwidth of bulk transfers. As opposed to other systems, the SHRIMP software architecture allows multiple users to access the resources of the network interface while providing protection. This is achieved through the mechanism of virtual mapped network interface into user space. The first prototype of SHRIMP was demonstrated on a cluster of Intel Pentium personal computers using the Intel Paragon routing backplane for interconnecting the cluster nodes. Custom designed network controllers are used for interfacing the cluster nodes to the network fabric.

Based on a study of a variety of parallel applications, Blumrich et al. (1994) concluded that the communication pattern of these applications is predominantly regular and predictable. Therefore, a mechanism for early binding of communication requests is introduced. The transfer of data is separated in two phases: setting up the transfer (map phase) and performing the data movement (send phase). During the map phase, a local buffer is mapped to a buffer residing in user space on a remote system. The setup phase involves the kernel for protection checking and storing mapping information, which results in a much higher overhead than in the second phase. This approach relies on the fact that once a transfer is set up, this transfer can be reused multiple times and the setup

overhead will be amortized over a large number of transactions. As a result, the performance optimizations are focused only on the activities related to the actual data transfers. This allows SHRIMP to reduce the effective system overhead of one message transfer to only a small number of instructions.

The communication model used by SHRIMP is distributed shared memory. By mapping local buffers to remote buffers, the communication is reduced to maintaining consistency of the two buffers. The underlying mechanism that enables this functionality is remote memory mapping. This mechanism maps segments of physical memory of remote nodes in the physical memory of the local node. These mappings are maintained in a network interface page table. The actual data transmission is performed through an automatic update operation. When the user process writes in a memory region that is mapped to a remote node's memory region, the automatic update mechanism is initiated. This mechanism is used to perform the memory coherency across the cluster. The actual update is triggered by the virtual mapped network interface that snoops the memory bus for writes.

Two modes of automatic update are provided – single and block mode. The single mode is optimized for minimal latency, while the block update is designed to achieve high bandwidth. The automatic update approach allows the CPU to initiate data transfers only through regular memory accesses. The only overhead the CPU incurs is associated with the local write-through cache latency. The actual transfer is later completed by the network interface, which allows for overlapping of communication and computation. Blumrich et al. (1996) extend the initial design of the virtually mapped SHRIMP network

interface to support user-level DMA transactions, which further reduce the software overhead for transmitting data between processors.

2.5.4 Fast Messages (FM)

Fast Messages (Pakin, Lauria, and Chien 1995) is a low-level messaging software system that aims to provide high-speed communication with low software overhead across a network of workstations. The major focus of FM is achieving low latency while preserving acceptable levels of effective throughput. FM recognizes that traditional software stacks impose high overheads to message processing and deliver only a small fraction of the physical capabilities of the network interconnects to user processes.

For demonstrating its concepts, the initial implementation of FM used a Myrinet network and presented a number of alternative designs through modifying the functionality of the MCP and the user-level software interface. In these designs, FM considered issues related to distribution of the communication workload between the central processor and the network coprocessor, buffer management, and scheduling of the activities on the host peripheral bus. FM addressed a number of challenges related to the limited capability of the LANai network processor to perform fast execution of the MCP whose processing is on the critical data path. Since the LANai processor is significantly slower than the main processor, FM has provided optimizations to reduce the number of cycles that LANai spends on each message.

FM presents a design space that offers a number of solutions for optimizing the trade-off between low latency and high bandwidth. Specifically, the use of programmed

I/O versus the use of the Myrinet NIC DMA engines is investigated. Programmed I/O uses the central processor to transfer the user buffers from main memory to the NIC local memory. This mechanism offers the lowest latency because it avoids the overhead associated with the DMA setup procedure and the synchronization between the main processors and the LANai. However, programmed I/O achieves lower effective bandwidth than the DMA approach. In order to combine the benefits of the two approaches, FM offers a hybrid scheme that uses programmed I/O at the sender and DMA transfers at the receiver.

The design considerations of FM are limited only to the trade-offs between latency and bandwidth. A performance analysis more relevant to practical situations might consider a multidimensional trade-off space formed by CPU overhead, overall application performance, latency, and bandwidth. In such a space, other communication schemes might offer a more balanced optimal solution.

2.5.5 Virtual Interface Architecture (VIA)

VIA is an industry-driven standard that specifies the interface between a high-performance system area network and a computer system (Compaq Computer Corporation, Intel Corporation, and Microsoft Corporation 1997). A major objective of VIA is to reduce the message-passing overhead and the number of intermediate data copies induced by general-purpose protocol stacks. This is achieved by collapsing the excessive number of software layers in traditional networking models, eliminating the

host operating system from the critical data path, and providing a hardware thread of control that is implemented by the VIA network interface controller.

VIA allows user processes to interact directly with the network controllers without mediation from the operating system. This avoids unnecessary context switches and significantly reduces the communication software overhead. Transfers to and from data buffers can continue even when processes participating in the communication are not scheduled for execution. Transferring data in and out of user memory when processes are swapped out is achieved through the NIC DMA engines and the VIA memory registration mechanism. This mechanism translates virtual addresses into physical addresses and pins user buffers in physical memory, similarly to U-net and SHRIMP. The registration mechanism guarantees that user data will remain in physical memory while the network controller accesses the buffers across the peripheral bus.

The major abstraction of VIA is the Virtual Interface (VI) communication channel. Each VI is a set of software and hardware mechanisms for synchronization and notification between the user processes and the NIC. The VI appears to user processes as an independent, fully functional network interface. Processes on remote nodes communicate between each other by creating a connection between their local VI instances. Each process can open multiple VIs. A VI cannot be shared between different processes. VI connections can be established by using both the client-server and the peer-to-peer models.

VIA is composed of three main components: the VI NIC, the VI Kernel Agent, and the VI User Agent (Figure 2.2). The VI Kernel Agent is implemented as a kernel-

level device driver performing operations that require operating system participation, such as: opening and closing the NIC device, obtaining system information, memory registration, creating VI instances, and establishing VI connections. The VI User Agent is a user-level library that implements the VI API, the VI queue interfaces, the notification mechanisms, the operation status report, and the error control (Compaq Computer Corporation, Intel Corporation, and Microsoft Corporation 1997).

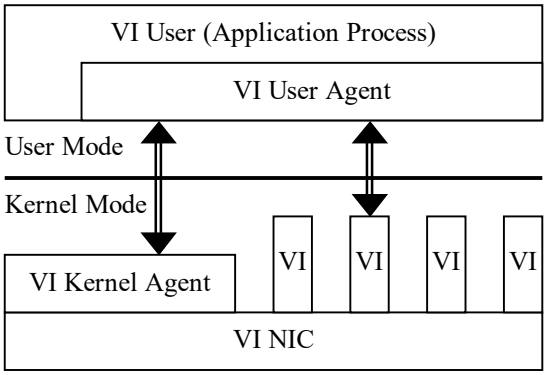


Figure 2.2 VI Architecture schematic

VIA network controllers are intelligent devices, usually implemented with ASIC and/or FPGA technologies. They are equipped with DMA engines capable of accessing system memory across the I/O bus without the participation of the system processor. These DMA engines can be used to transfer data directly from the user buffers to the network fabric, and vice versa, avoiding intermediate copies in host system memory. This increases the effective bandwidth of communication, frees the main CPU from communication tasks, and reduces congestion over the system bus. DMA engines support scatter/gather mode of operation, which can be used for efficient zero-copy transfers of

noncontiguous memory layouts. VIA controllers are designed to have sufficient resources to support a large number of VI instances. Data structures associated with each VI instance are maintained in NIC memory, which allows NIC's to process requests for message transfers faster and with reduced control traffic over the I/O bus.

The interface between the user processes and the VIA NIC is implemented through a pair of send and receive descriptor queues associated with each VI instance. The service policy of the queues is FIFO. Using these queues, the VIA NIC keeps track of the send and receive requests and also maintains their correct order. The VI descriptors are data structures that reside in user space and specify the source and/or target data buffers, the desired operation, and the method of completion notification. User processes allocate, initialize, maintain, and free the descriptors. The descriptors are accessed and interpreted by VIA NIC. This is achieved by pinning the memory segments that contain the descriptors to the physical memory. This is performed by the VI memory registration mechanism. After users allocate and initialize a descriptor, they invoke a library call from the VI User Agent to notify the NIC about the requested operation. This notification occurs through updating the VI doorbell. Doorbells are data structures that reside in NIC local memory and are mapped into the address space of the user process. Thus, by accessing certain memory regions in its memory space, a process can interact with the NIC. These accesses are mediated by the VI User Agent.

VIA provides synchronous and asynchronous methods for completion notification. The synchronous method is implemented by polling a flag in the NIC's memory. This flag is signaled by the NIC processing element when the requested

operation is completed. The asynchronous method uses interrupts for notification. When the NIC completes the operation, an interrupt to the main processor is generated. This interrupt is serviced by the VI Kernel Agent, which in turn signals the user process through an operating system synchronization mechanism, such as an event in Windows or a conditional variable in Linux. The polling method offers lower latency than the interrupt method. However, the polling mode engages the host processor and causes high CPU overhead, which to a large degree defeats the purpose of the complex architecture of the VIA NIC. Polling can be used efficiently for sending short messages whose total transmission time is comparable to the time necessary for interrupt processing.

In order to enable users to specify their buffers with virtual addresses, VIA provides a memory registration mechanism. This mechanism pins virtual memory segments participating in data transfers to the physical memory. This allows the DMA engines to access data buffers at any given time without the participation of the operating system, even when the user process is swapped out. This feature improves communication efficiency and minimizes processor overhead. In addition, memory registration increases communication predictability, which is a feature that can be successfully used in systems with strict timing requirements. Specifically, pinning eliminates the variability of the operating system's virtual memory mechanism. MPI/RT (Kanevsky, Skjellum, and Rounbehler 1998) is an instance of a message-passing specification with real-time requirements that may benefit from VIA memory registration and the VI quality of service attributes that are proposed for the next version of the VIA specification.

VIA defines remote DMA (RDMA) Read and Write operations for transferring data from one network node to another, transparently to the main processor of the two computer systems involved in communication. In an RDMA read operation, the requesting process instructs the local NIC about the remote node, the remote buffer, and the desired RDMA operation. Then, the local NIC initiates the transfer by contacting the remote NIC without involving the remote processor. Using its DMA engine, the remote NIC copies the remote user buffer into its memory and then transmits it over the network to the NIC on the system that requested the operation. Finally, the NIC local to the requestor performs a local DMA transfer to deposit the content of the remote buffer into the host memory and notifies the requesting processes about the completion of the operation.

The RDMA mechanism relies on the implementation of a cache-coherence protocol on the host computer systems. This protocol maintains the consistency of the cached memory segments that are modified transparently to the main CPU by the DMA engines on the VIA NIC. The RDMA read and write methods can be successfully used for implementing *put* and *get* primitives on distributed shared-memory systems as well as the MPI-2 one-sided communication model (Message Passing Interface Forum 1998).

The direct access of user processes to network resources, and especially to the NIC DMA engines that can access the entire physical address space of the host system, raises certain protection and security concerns (Dimitrov and Gleeson 1998). Thin communications software stacks facilitate low-overhead data transfers but, at the same time, give users high privileges for accessing and manipulating system hardware

resources without the mediation of the host operating system. Targeting these protection and security concerns, the VIA specification provides a set of mechanisms, such as the VI memory model and the VI connection model, that increase the level of protection. VIA isolates user processes on the same system from each other and also protects the operating system from malfunctioning or malicious user processes. In addition, VIA provides mechanisms for effective monitoring and control of VI connection creation.

Direct access of user processes to NIC resources, NIC DMA engines, remote DMA operations, and the VI memory management model are among the major facilitators of low-overhead communication with zero-copy transfers. Most of these mechanisms have been introduced by network technologies such as Myrinet and numerous research projects among which most notably are U-net and SHRIMP (von Eicken and Vogels 1998). VIA is an important step toward creating a uniform interface view of high-speed networks with user level access to communication hardware. Underlying details of data-link and physical network layers remain hidden for the upper communication software layers. This significantly increases portability of message-passing systems (or directly coded VIA applications) while preserving optimal performance. In addition, VIA provides mechanisms for increased protection and security, which are important factors for production systems. Several networking vendors, such as Giganet and Compaq provide PCI VIA implementations for Windows and Linux. These implementations are based on diverse vendor-specific physical layers. Their compliance with the VIA specification allows a message-passing system, such as MPI, to be ported quickly from one network to another with minimal code changes while

preserving optimal performance. The portability characteristics of VIA are successfully demonstrated by the MPI implementation presented in this work. This implementation provides support for Giganet and ServerNet for Windows and Linux with minimal changes. These changes are mainly in host name resolution and VI connection creation.

2.6 Message-Passing Interface

The Message-Passing Interface (MPI) has become a de-facto standard for programming multicomputers and multiprocessor architectures as well as clusters of workstations. The standardization effort of MPI was initiated by United States government agencies and embraced by high-performance computing vendors and users relying on parallel processing for intensive scientific computations. MPI builds on the experience of other message-passing systems, such as PVM (Geist et al. 1994) and Zipcode (Skjellum et al. 1994). Among the major goals of MPI is creating a portable interface for parallel programming while providing maximum performance. MPI offers a unique combination of abstraction power, performance, and portability. These features have all contributed to the success of MPI as the parallel API of choice in the past several years. Recently, the audience of MPI has expanded from the traditional scientific computing area to the rapidly growing area of cluster computing.

MPI provides a unique opportunity to parallel application programmers to create portable algorithms that can be easily moved to different platforms while also achieve portable performance. The performance portability feature is based on the MPI portable interface that can be optimized on different target architectures to reflect features and

capabilities specific only to these architectures. If such optimizations are directly encoded in the application software, it is likely that their performance will not be portable. This fact often attracts less attention than the code portability, but has an important impact on the economics of high-performance computing software by providing opportunities for optimizations that are not tied to a single platform.

The success of MPI has triggered specifications for extensions to the original standard, which lead to the creation of MPI-2 and MPI/RT. MPI-2 offers much wider functionality than MPI, including parallel file I/O, extended collective operations, one-sided communication, and dynamic process management (Message Passing Interface Forum 1998). Dynamic process management addresses the limitation of MPI related to the static model for allocating processes. MPI-2 allows the MPI jobs to grow by dynamically spawning new processes. A number of self load-balanced algorithms can benefit from this new functionality. Dynamic task allocation also addresses environments with dynamically changing rates of data input.

MPI/RT (Kanevsky, Skjellum, and Rounbehler 1998) emphasizes predictability, scheduling, and early binding of communication activities. MPI/RT introduces the notion of explicit communication channels that are created on demand by user processes. This differs from the MPI paradigm, which mandates virtual all-to-all connectivity and allows every process to communicate with any other process. Resource management is another area in which MPI/RT provides explicit control. User processes can manipulate buffer pools and select policies for the order of buffer processing. Common targets for MPI/RT are embedded systems that perform time-critical computations and intensive real-time

processing on data coming from external devices, such as radar, sonar, or other remote sensors. VIA-based targets for video on demand or other purposes are plausible.

2.7 Review of Approaches for Improving Parallel Performance

The approaches to improving performance of parallel systems can be generally divided into four major groups: improving computational capabilities of network nodes, increasing the speed and scalability of network interconnects, optimizing communication system software, and optimizing parallel application algorithms. Among the hardware approaches are optimizing microprocessor architectures for higher instruction and data throughput, adding vector units to superscalar cores such as G4 AltiVec and Intel SSE, increasing the clock rate, improving the throughput of system busses, improving memory hierarchies, creating faster and wider peripheral busses, and providing new methods for faster I/O, such as InfiniBand (Intel Corporation 2000) and RapidIO (Motorola 2000). Some of the approaches for improving network interconnects are increasing the physical data link rates (presently at 1-2 Gbit/sec), increasing processing capabilities of network interface adapters with larger memory and faster processor components, improving packet switching and routing, and providing scalable interconnect topologies that enable clusters with thousands of nodes.

The attention of this work is focused on the system software approaches and their interactions with the network resources and the parallel application algorithms. Earlier in this chapter, the user-level networking approach to communications software was reviewed. This approach is considered one of the most significant factors for enabling

clusters for high-performance computing. Different variations of user-level networking are commonly used in today's computational clusters, most notably VIA and GM – the software interface of Myrinet (Myricom 2001). Another common software approach to improving parallel performance is optimizing collective communication operations. A large group of data parallel applications use collective operations for performing message passing; therefore, improving performance of these operations can lead to a significant improvement in overall performance. A number of widely used message-passing systems, among which is MPICH, provide minimal or no optimizations for collective operations. For this reason, a large group of applications are created using only point-to-point communication primitives, and not the collective operations. This is a shortcoming of a parallel algorithm's architecture, because it limits the capability of applications to adapt their performance to the specific characteristics of new target systems. These characteristics are often exploited by the native collective operations.

Minimizing the number of transfers, reducing network contention through scheduling, minimizing traffic on slow links, and increasing the degree of concurrency of individual transfers are among the common approaches for optimizing collective operations. These approaches have an important role in improving overall performance and scalability. One of the frequently used techniques for optimizing collective operations is employing algorithms with lower than linear asymptotic complexity, such as binary trees, minimum spanning trees, and fat trees. The execution time asymptotic complexity of these algorithms is $O(\log p)$. Although they result in the same number of transfers as linear algorithms, the tree algorithms provide opportunities for greater

concurrency and better utilization of the available bisection bandwidth. The role of asymptotic optimizations is especially important on large-scale systems with hundreds to thousands of nodes.

Barnett et al. (1994) provide a different perspective on optimizing collective operations. They study the trade-off between asymptotic complexity and overall amount of exchanged data. They recognize the importance of concurrency and that practical message-passing systems usually provide multi-protocol implementations of data transfer based on message length. As an example of the approach applied by Barnett et al. (1994), the *MPI_Reduce_scatter* operation is reviewed. This MPI call performs a reduction operation on the input buffers with size L bytes of all participating processes and then scatters the result among these processes. Typically, asymptotically optimal algorithms perform this operation in two phases: first the reduction and then the scatter operation. Each of these phases can be implemented as a $\log p$ tree, each of which requires $p - 1$ transfers, for a total of $2\log p$ tree stages and $2p - 2$ transfers. All of the transfers of the reduction operation are of the same size L. The receive buffers of the scatter phase are of size L/P. The scatter transfers at stage j in the tree are of size $L/(2j)$, where j is in the range $[1, \log p]$. The total amount of data moved with the asymptotically optimal implementation is $(p - 1)L$ for the reduce phase and $\log p(L/2)$ for the scatter stage, totaling $(p - 1)L + \log p(L/2)$. Barnett et al. recognize that although the $\log p$ algorithms are optimal in terms of number of transfers and asymptotic complexity, they exhibit limited concurrency – half of all transfers are performed during one stage and the other half in all $\log p - 1$ stages. Barnett et al. (1994) propose a bucket class of collective

algorithms that addresses concurrency of transfers, network contention, and the trade-off between algorithm complexity and total amount of exchanged data. The proposed algorithm that implements the operation equivalent to *MPI_Reduce_scatter* is "bucket distributed global combine." This algorithm requires $p - 1$ communication steps and moves $(p - 1)L$ bytes of data in $p(p - 1)$ transfers. On one hand, the proposed algorithm has higher complexity than the asymptotically optimal algorithm, since $(p - 1) > 2\log p$ for any $p > 6$. Also, the bucket algorithm uses a larger number of transfers $p(p - 1)$ versus $2p - 2$ for the asymptotically optimal algorithm. On another hand, the bucket algorithm exchanges a smaller total amount of data with $\log p(L/2)$ bytes, while engaging all nodes in communication and computation in every stage. As a result, the communication and computation loads are more evenly distributed among all nodes. Barnett et al. (1994) show that the bucket algorithm performs better than the asymptotically optimal algorithms on a number of parallel platforms.

Another software approach to improving performance on parallel systems is to reflect network topology and connection attributes in the communication software. Baum et al. (1998) demonstrate an MPI implementation based on MPICH with explicit support for both network and shared-memory communication. Baum et al. (1998) have implemented collective algorithms that are tuned to reflect the mapping of process ranks to processors and also the difference in communication speed of the two MPI devices. The main objective of the optimization is to minimize the number of slower network transfers and to reduce network contention by scheduling transfers over the same network links. The results of this study show that the completion time of collective algorithms can

be significantly reduced (more than two times for broadcast) when the proposed topology-aware algorithms are applied to a cluster of multi-processor nodes. The study presented by Baum et al. (1998) is limited to the case when the parallel system supports one slower and one faster device. In order to meet the needs of practical systems, this research can be extended to cover hierarchical communication topologies with an arbitrary number of devices and dynamically evaluated performance attributes. There is already demand for such research since clusters with multi-processor nodes and two or more different networks are presently being built. A typical example for such a configuration is an organization that operates two kinds of high-speed network clusters and would like to execute parallel jobs across all nodes. This can be achieved only by using standard TCP communication when nodes from the different clusters communicate. In this scenario, the MPI middleware must support at least two networking devices (a high-speed network device and a TCP device) as well as one shared memory device in order to facilitate efficient processing on the aggregate cluster.

Early binding is another important software mechanism that can lead to significant parallel performance improvement. In message-passing systems, early binding is used for exploiting temporal locality, specifically when the same data buffer can be reused for multiple transfers. According to their data distribution, parallel algorithms can be classified as *structured* and *unstructured*. The structured algorithms often exhibit static data distribution, which is maintained during the entire duration of the algorithm. According to their temporal behavior, parallel algorithms can be classified as *regular* and *irregular*. Regular algorithms have repetitive communication pattern, which typically

results in a loop that involves the same data buffers updated at each loop. Structured and regular parallel algorithms are the best candidates for optimizations that use early binding. Typical representatives of structured and regular algorithms are iterative solvers for systems of linear equations.

Early binding allows message transfers to be set in the initialization phase of the algorithm and then reused multiple times during the communication intensive portions of the algorithm. Thus, the initialization overhead associated with message setup can be amortized over a large number of messages, which increases the effective message passing performance. The MPI standard (Message Passing Interface Forum 1994) specifies a persistent mode of communication, which allows user processes to allocate persistent send and receive point-to-point requests that can be reused multiple times. Skjellum (1998) extends the idea of persistent requests and proposes a persistent model for collective communication as a performance extension to MPI. This model enables algorithms that utilize MPI collective operations to take advantage of early binding and minimize the effective cost of communication. The collective operations specified in the MPI and MPI-2 standards do not allow for such performance optimizations. In contrast, MPI/RT (Kanevsky, Skjellum, and Rounbehler 1998) with its collective channel abstraction does support such optimizations.

The last software mechanism for improving parallel performance reviewed in this chapter is overlapping of communication and computation. This mechanism demonstrates the complex interactions of the hardware and software components of a parallel system. Effective overlapping is possible if and only if the entire software stack and the

underlying hardware are implemented with awareness of this mechanism. Each layer of the communication stack is equally important for delivering the capabilities of overlapping to application software. At best, upper layers can preserve these capabilities. A critical characteristic of system software and message-passing middleware is to propagate these capabilities with minimal losses. Only then, can an application using overlapping actually achieve real performance gain. If the underlying layers do not support overlapping, even the most optimally implemented algorithm will not be able to produce any performance improvement.

Tanaka et al. (1998) and Baden and Fink (1998) study the performance effect of overlapping on clusters of SMP machines and present models for utilizing overlapping on the studied systems. The model proposed by Baden and Fink targets hierarchical multicomputers and uses a non-uniform distribution of the workload. This model supports a node-level, rather than processor-level, communication and designates one process per node to participate in communication. Only processes on the same node communicate directly. Inter-node communication is performed by the designated communication processes.

The main objective of this model is to achieve overlapping by over-decomposing the problem size using a larger number of processes than processors. This approach relies on the operating system to schedule a process that can perform computation while another process is blocked on a communication event. The model proposed by Baden and Fink (1998) is non-homogeneous and presents a high-degree of complexity when a parallel algorithm is designed. Also, the actual behavior depends on operating system

characteristics, which can vary greatly from one operating system to another or even on the same operating system but on different architectures. These characteristics limit the applicability and portability of this model. Further, Baden and Fink do not study the actual effectiveness of overlapping. They rely on the fact that the operating system will schedule a process that is independent of a communication event while another process is blocked on such an event. However, they do not show if the communication activity scheduled by the communicating process is actually executed concurrently with the processing performed by the computing process or whether these activities are in fact performed sequentially. Although the study by Baden and Fink (1998) provides a valuable methodology for addressing performance on SMP clusters, this study shifts its focus to issues related to data decomposition and operating system scheduling. Other issues that are critical for achieving effective overlapping but not discussed by Baden and Fink (1998) are as follows: capability of the underlying hardware to perform independent communication, sufficient memory bandwidth, CPU overhead, adequate support for overlapping from the system software, and scheduling of communication and computation within the same process.

The model proposed by Tanaka et al. (1998) is demonstrated on a cluster of workstations interconnected with Myrinet. A low-overhead communication layer with distributed shared-memory primitives is implemented to perform data movement between nodes. Intra-node communication between processing threads is based on shared memory. The communication layer implements remote memory operations and synchronization primitives, which utilize the processing capabilities of the Myrinet

adapters. This model allows for homogeneous distribution of data among processes. The model relies on an explicit use of multiple threads per processor for achieving overlapping. Similarly to the effort of Baden and Fink (1998), the work presented by Tanaka et. al (1998) does not provide an in-depth systematic study on the factors that influence overlapping.

Sohn et al. (1999) investigate overlapping from the standpoint of the parallel algorithms. Sohn et al. recognize that algorithms that can potentially take advantage of overlapping have similar architecture. These algorithms are based on processing with two distinct phases – a local computation phase and a communication phase. These phases can be repeated in a loop. The traditional approach is to perform these two phases sequentially, thus eliminating the possibility for overlapping. Sohn et al. observe that a generalized approach to using overlapping can be applied to these algorithms. The communication and computation phases can be further subdivided into smaller fragments, which are interleaved and pipelined. Thus, instead of exchanging one message of size L, the communication phase exchanges a series of s segments of size m, such that $L = s\ m$. In case the algorithm is computationally bound, Sohn et al. (1999) claim that by selecting an appropriate segment size, it is theoretically possible to hide the entire communication time. Ideally, the larger the number of segments m is, the more effective the overlapping becomes. However, by applying theoretical models such as LogP and LogGP as well as performing experimental measurements, Sohn et al. observe that the effectiveness of overlapping is limited by the efficiency of small message transfers for which overhead becomes the predominant component of the transmission time. They

define a metric, called communication efficiency, as the ratio $(T_{comm,1} - T_{comm,s}) / T_{com,1}$ where $T_{com,1}$ is the time for transmitting a message of size L in one segment and $T_{comm,s}$ is the time for transmitting the same message in s segments of size L/s. Sohn et al. use communication efficiency to evaluate the capability of a parallel system to perform overlapping of communication and computation.

Of the above reviewed research efforts on overlapping, Sohn et al. present the most elaborate study. They investigate the impact of the structure of parallel algorithms that use overlapping on the communication pattern of these algorithms. The rest of the studies are limited to implementing certain algorithms using overlapping and presenting a summary of results on practical systems. Sohn et al. go a step further by studying the effects of overlapping on algorithms and the trade-offs between overlapped communication and computation with a different number of message segments. However, Sohn et al. take a restricted view of the capability of a system to achieve overlapping. The basis for identifying this capability is the "communication efficiency" metric. Although this metric reveals certain insights about the trade-offs between message size and number of segments, it mainly emphasizes the capability of a system to improve its communication efficiency by pipelining multiple messages and thus reducing the aggregate impact of message overhead. As the experimental results presented in Sohn et al (1999) show, the communication efficiency metric penalizes parallel architectures with highly optimized network fabrics, such as Cray T3E, because the communication overhead of these systems is insignificant and pipelining has little to hide. By its

definition, the “communication efficiency” metric presents only a restricted view of the multi-faceted issue of effective overlapping.

None of the models for overlapping reviewed in this section investigate the importance of processor overhead, the impact of hardware and system software, and the interaction between the communication subsystem and the application process in sufficient depth. These factors play a critical role in understanding the complex processes in a parallel system and in designing parallel system and application software that utilize overlapping effectively. The studies presented here have not offered sufficient formal description of overlapping and its incorporation in theoretical models for parallel computing. The practical systems suggested by the research are often highly customized and do not offer significant benefit to the wide user base of parallel processing, which primarily uses standard message passing interfaces, such as MPI. Furthermore, these studies have not paid sufficient attention to the conditions that enable effective overlapping. These conditions include:

- sufficient bandwidth of memory subsystem,
- sufficient bandwidth of peripheral I/O bus,
- host bus-master capabilities of network controllers,
- asynchronous message completion notification,
- independent message progress, and
- low CPU overhead.

The goal of this work is to address these limitations by providing an in-depth study of overlapping by:

- presenting a formal definition and theoretical description,
- incorporating overlapping in models for parallel computation,
- studying the factors that affect overlapping,
- defining new metrics that reveal interactions critical for effective overlapping,
- demonstrating a new MPI implementation that emphasizes overlapping,
- implementing well-known algorithms to take advantage of overlapping,
- evaluating the performance improvement achieved through the use overlapping, and
- formulating guidelines for communication system hardware and software designers and algorithm developers.

2.8 Conclusions

This chapter reviewed fundamental concepts of parallel performance analysis and important models for parallel computing in distributed environments. These concepts and models are used throughout this work to lay the theoretical basis of its hypothesis and to demonstrate the performance gain of the optimizations presented here. Then, the main characteristics of parallel architectures were outlined, and parallels and contrasts between massively parallel processors and clusters were made. Further, this chapter focused on high-speed networks as a critical component of computation clusters, and the concepts of user-level networking and low-overhead messaging layers were reviewed. By reducing message-passing overhead and eliminating intermediate data copies, efficient software interfaces become an important contributor to achieving high-performance on clusters.

Next, a review of MPI and related message-passing specifications was presented. MPI has become the most widely used interface for portable programming of parallel computers. The ability of MPI and the underlying software layers to propagate hardware capabilities to user applications is a critical requirement for accomplishing scalable parallel processing though exploiting the mechanisms studied in this work. Finally, this chapter presented a summary of software mechanisms for improving parallel performance. These mechanisms should be carefully designed and implemented in the entire stack of system and application software, as well as appropriately supported by the hardware, in order to deliver the expected performance gains.

CHAPTER III

A THEORETICAL FRAMEWORK FOR EARLY BINDING AND OVERLAPPING OF COMMUNICATION AND COMPUTATION

3.1 Objectives and Constraints

The framework presented here offers a description of the communication and computation processes in a high-performance parallel system while also accounting for the effects of early binding and overlapping of communication and computation. The focus of the framework is on distributed parallel environments with high-speed networks and is restricted to those using message passing for communication. Although some of the results and conclusions presented in this framework can be applied to a broader class of parallel systems, it is likely that for achieving valid results on these systems, some of the assumptions made here may need to be adjusted to the specific target environments.

This theoretical framework focuses on the interaction between the application processes and the message-passing middleware. One of the distinguishing characteristics of the framework is that it emphasizes the application's point of view. This is accomplished by hiding specific architectural and performance details of the hardware

and low-level system software. The performance parameters presented in the framework and used by the models are those observed by the parallel application, not those that can potentially be achieved by the underlying computer and communication infrastructure. This view allows for a more realistic description of the complex interactions between the hardware and software components of the parallel system. Specific attention is paid to the influence of the middleware providing message-passing capabilities, specifically MPI. The message-passing middleware and its effects on performance are often insufficiently studied in parallel computing models. The framework presented in this chapter attempts to address this insufficiency, and will be used practically to reveal the importance of the architectural of the MPI implementation presented in Chapter IV.

3.2 Parallel Computation Model

This section presents a parallel computation model that meets the objectives of the theoretical framework. In this study, this model will be referred to as BOUM – "Bandwidth and Overhead [based parallel processing] User [level] Model." Some of fundamental parallel programming models that have received wide-acceptance were reviewed in Chapter II. Specifically, the attention was focused on the BSP (Valiant 1990) and LogP (Culler et al. 1996) models. BOUM incorporates features of both of BSP and LogP. The justification for using BOUM is the insufficiency of these models for the purposes of the proposed framework. Below, the requirements for this model are stated while the insufficiency of BSP and LogP is also noted.

3.2.1 Requirements

The first requirement of the target model is that it should enable a quantitative description of overlapping of communication and computation. A prerequisite for this requirement is the explicit modeling of both communication and computation. The models for parallel programming studied earlier can be divided into two categories: those that model both communication and computation and those that model communication only. For example, BSP (Valiant 1990) accounts for both communication and computation, while LogP models communication only (Culler et al. 1996). The second requirement of the target model should be to describe asynchronous processing. Asynchronous communication is an important requirement for efficient overlapping. BSP is a bulk-synchronous model and does not provide for fine granularity of pair-wise asynchronous activities. Next, the target model should account for messages with varying frequency and sizes. Parallel implementations of numerical algorithms used in high-performance computing can result in a variety of communication patterns and message lengths. By way of comparison, LogP emphasizes applications that predominantly use small messages (a few bytes to a few hundred bytes).

Finally, the target model should offer a parameter that represents message overheads as experienced by the parallel applications. The proposed framework will use the overhead parameter to express the benefits of early binding. Again, for comparison's sake, LogP uses an overhead parameter, while BSP does not. This subsection stated a number of requirements that a parallel computing model will need to meet in order to be successfully used for the analysis necessary for achieving the goals of this dissertation.

Widely used models in the theory and practice of parallel processing, such as BSP and LogP, provide only partial support for these requirements. The next subsection defines a model that meets all requirements.

3.2.2 Statement of Model and Definition of Parameters

BOUM, the model being introduced here, expresses the parallel execution time, T_p from standpoint of an individual process that participates in the parallel job as a sum of its computation time T_{comp} and communication time T_c:

$$T_p = T_{comp} + T_c \tag{3.1}$$

The unit for the parallel execution, computation, and communication times is seconds. The computation time is further expressed as follows:

$$T_{comp} = F(W)t_c, \tag{3.2}$$

where, $F(W)$ is a function of the problem size W, expressed as a number of basic compute operations, such as floating point additions or multiplies, and t_c is the time for execution of one of these basic operations. For example, the compute time for the addition of two vectors of n floating point elements would be given by $T_{comp} = nt_c$.

The communication time T_c is the sum of the communication times of all messages exchanged by a process. The communication time of a single message of size m is in turn expressed as a sum of two components – overhead o and transmission time bm:

$$T_c(m) = o + bm, \tag{3.3}$$

where, b is the inverse of the bandwidth as experienced by the user process, measured in seconds per byte [sec/byte]. The overhead parameter o is similar to the overhead

parameter used in the LogP model (Culler et al. 1996) and the bandwidth factor *b* is similar to the parameter *G* in the LogGP model (Alexandrov et al. 1995), where *G* represents the gap between the bytes of a long message. As opposed to these models, BOUM uses parameters that are measured with respect to the user process (*i.e.*, they are directly observable by the applications). Thus, the model abstracts platform's hardware details (*e.g.*, processor clock rate and network physical bit rate) and provides a more generalized description of performance while preserving representation accuracy. Another advantage of BOUM's approach model is that users can measure the parameters of the model by writing simple MPI programs. Then, these values are replaced in the execution time derivations obtained with the model for estimating performance.

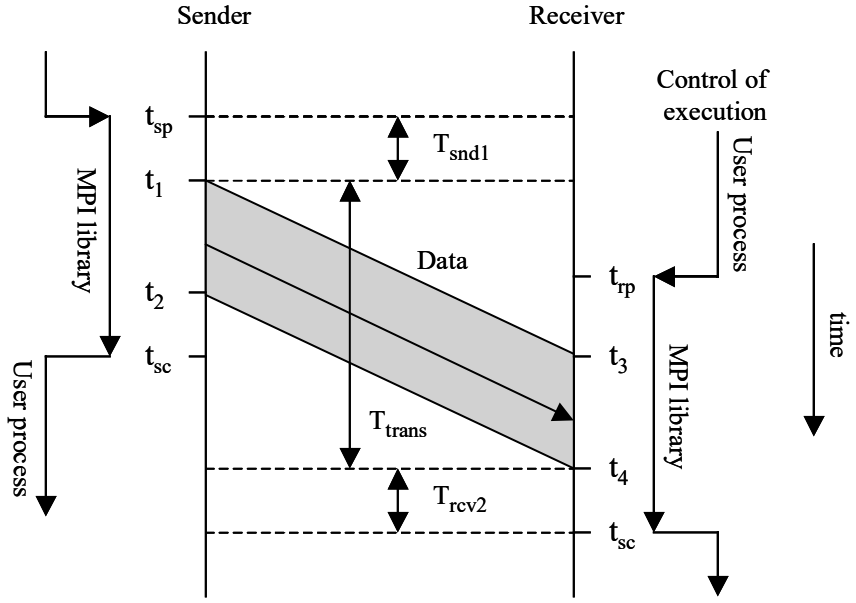


Figure 3.1 Overhead and transmission time

The overhead component o of the communication time of a message of size m can be expressed as a sum of the T_{snd1} and T_{rcv2} components of the send and receive overheads respectively, as depicted in Figure 3.1 (*i.e.*, $o \equiv T_{snd1} + T_{rcv2}$). Then,

$$T_c(m) = T_{snd1} + T_{trans} + T_{rcv2} = o + bm, \tag{3.4}$$

where $T_{trans} = bm$. Figure 3.1 represents the activities that are used for definition of the send and receive overheads as well as the definition of the transmission time. The send overhead has two components. The first component is the period between the time when the user process posts its send request t_{sp} and the time when the first byte of the message is placed on the network t_1. The first component of the send overhead is denoted by T_{snd1}. The second component of the send overhead T_{snd2} is the period between the time when the last byte of the message is placed on the network t_2 and the time when the send process is notified about the completion of the send request t_{sc} (*i.e.*, $T_{snd2} = t_{sc} - t_2$). In some systems, depending on the capabilities of the transport, the moment in time t_{sc} may be earlier than t_2. However, the relationship between t_{sc} and t_2 does not change the definition of the total message overhead because it is assumed that T_{snd2} is overlapped with the sum of the transmission time $T_{trans} = t_4 - t_1$ and the second component of the receive overhead T_{rcv2}; t_4 is the time when the last byte of the message is deposited in receiver's memory. Theoretically, it is possible that $T_{snd2} > T_{trans} + T_{rcv2}$. However, in most practical systems, the completion notification overhead of both the send and receive requests is either approximately the same ($T_{snd2} \approx T_{rcv2}$), or the send notification overhead is smaller ($T_{snd2} < T_{rcv2}$). Therefore, for these systems, we find that $T_{snd2} < T_{trans} + T_{rcv2}$

because $T_{trans} > 0$, which justifies the assumption that T_{snd2} can be overlapped with the transmission time T_{trans} and the notification overhead of the receive request T_{rcv2}.

The definition of the receive overhead is similar to the definition of the send overhead. The first component of the receive overhead T_{rcv1} is the period of time between the moment when the user posts the receive request t_{rp} and the moment when the first byte of message arrives at the destination node t_3. The second component T_{rcv2} is the period of time between the moment when the last byte of the message arrives t_4 and the moment when the user process is notified about the completion of the request t_{rc}. Similarly to T_{snd2}, it is assumed that for practical purposes T_{rcv1} is overlapped with T_{snd1} + T_{trans}, and hence is not included in the overhead parameter of the model. It is possible that the user process can post an asynchronous non-blocking receive request (by calling *MPI_Irecv*, for instance) earlier than t_{sp} (*i.e.*, the receive request is posted before the send operations is started). In this case, clearly $T_{rcv1} > T_{snd1} + T_{trans}$, so the first component of the receive T_{rcv1} overhead cannot be fully overlapped with the first component of the send overhead T_{snd1} and the transmission time T_{trans} as suggested by Figure 3.1.

It should be noted that the execution line of control (shown in Figure 3.1) assumes that both the send and receive requests are blocking. Here, a distinction between "blocking completion notification" and "blocking [semantics of communication] requests" should be made. As mentioned earlier, the former use of the term "blocking" is related to the synchronization procedure between the user process and the message-passing library. The options according to completion notification are polling and blocking. The latter use of the term "blocking" is related to the transfer of the flow of

control between the user process and the message-passing library when a communication request created by the user process is submitted for execution to the library. The options according to the transfer of control in request submission are blocking and non-blocking Blocking requests can be implemented with both polling and blocking synchronization. Similarly, blocking notification can be used by both blocking and non-blocking requests.

In blocking mode, the control of execution is transferred from the user process to the communication library from the moment when the request is posted, until the moment when the request is completed, according to the completion semantics of the particular message-passing API. The blocking mode of communication limits the opportunities for overlapping and early binding. In contrast, the non-blocking and persistent modes (Message Passing Interface Forum 1994) facilitate efficient overlapping and early binding. The following discussion on hiding and shifting the send and receive overheads and the message transmission time assumes that the non-blocking mode is used. Using the MPI API, the send and receive requests can be posted with *MPI_Isend* and *MPI_Irecv* respectively, and their completion checked at a later moment with *MPI_Wait* or *MPI_Test*. One of the most important characteristics of the non-blocking communication is that it allows user processes to schedule message overhead and transmission time according to some strategy for improving application performance. This strategy can utilize such mechanisms as hiding overhead, message pipelining, and overlapping.

Once the non-blocking receive request is posted, the message-passing middleware typically does not perform any processing on this request until a matching message arrives (*i.e.*, until t_3). According to this scenario, T_{rcv1} will simply be shifted in time and,

as a result, the application can perform other computation or communication in the period between the moment when the request is posted and t_3. Therefore, this shift of the moment of request posting will not result in an effective overhead increase. In fact, this behavior is encouraged by most MPI implementations because the first component of the receive overhead can be shifted to earlier parts of the parallel algorithm that are not overhead sensitive (*e.g.*, in the initialization phase of the algorithm). Then, instead of overlapping the send overhead T_{snd1} and the message transmission time T_{trans} with the first component of its receive overhead T_{rcv1}, the receive process can overlap the period of time $T_{snd1} + T_{trans}$ with useful computation or other communication. This will decrease the effective receive overhead incurred by the user process and improve the opportunities for overlapping of communication and computation. These opportunities are further enhanced by message-passing middleware that supports independent message progress. If such a service is available, the user process may delay the completion synchronization (*e.g.*, *MPI_Wait*) until a moment after t_4, which would enable this process to effectively overlap the entire transition time with other activities.

The send process can achieve effective overlapping of communication and computation, similarly to the receive process, by shifting the synchronization procedure for completion of the send request to a later moment, and scheduling computation activities immediately after the send request is registered with the MPI library (*i.e.*, after t_1). This can effectively hide the transmission time (assuming sufficient memory bandwidth) and also move the notification overhead to a non-time-critical segment of the algorithm. The actual benefit of overlapping depends on the capabilities of the computer

platform, the network infrastructure, and the communication software. A main objective of this study is to reveal the factors that affect overlapping, how overlapping efficiency can be improved, and how parallel algorithms can take advantage of overlapping.

In summary, the proposed BOUM model uses three parameters for describing communication and computation. These parameters are as follows:

- t_c [sec] – time for basic unit computation operation,
- o [sec] – message passing overhead,
- b [sec/byte] – inverse of the effective bandwidth.

Using these parameters, the parallel execution time can be expressed as follows:

$$T_p = F(W)t_c + \sum_{i=1}^{N}(o + bm_i) \tag{3.5}$$

where, $F(W)$ is the problem size, N is the number of message transmissions performed by the modeled process and m_i is the size of the ith message. Expression (3.5) makes an initial assumption that the message overhead o is constant for all messages, regardless of their size. Later in Chapter III, it is shown that this assumption is too optimistic for many practical cases and is further refined.

3.2.3 Accuracy of Description

The BOUM model presented here achieves a realistic description of communication because it accounts for a number of implementation-specific features of the message-passing middleware that affect the effective communication performance delivered to the user applications. Among these features are as follows:

- optimizations for minimum latency of short messages, which often lead to additional data copies
- specifically designed protocols for long message exchanges – typically, three-way rendezvous protocols,
- packet headers associated with control information,
- control messages used for user-level flow control or other purposes,
- mechanisms for message completion notification, and
- type of progress engine of the message-passing middleware[1].

As opposed to BOUM, other parallel computation models that use the raw network parameters omit these important performance-defining factors. Furthermore, the application view of BOUM accounts for hardware limitations, such as peripheral bus saturation or issues related to the multiprocessor architecture of the cluster nodes as well as the impact of the host operating system. For instance, the cost of the process and thread context switches could substantially change the real values of the message overhead that application processes experience. Experimental results show that a process context switch on a machine with more processors is longer than on a machine with one processor, and that a thread context switch on Linux is substantially longer than it is on Windows. These differences can be significant, as demonstrated in Chapter IV.

This section introduced BOUM, a parallel computation model used for theoretical description and parameterization of early binding and overlapping of communication and

---

[1] A detailed description of the MPI message progress engine is presented in Chapter IV.

computation as fundamental software mechanisms for improving parallel performance on clusters of workstations interconnected with high-speed networks. BOUM is based on widely used models such as BSP, LogP, and LogGP but makes important distinctions, specifically it,

- uses parameters that are observable by the user applications,
- provides for asynchronous processing,
- explicitly describes both communication and computation, and finally
- enables parameterization of overlapping and early binding.

In the following sections of this chapter, BOUM is applied to describe early binding and overlapping of communication and computation. Selected parallel algorithms are analyzed and the performance gains from overlapping and early binding are estimated using this model. These estimations are validated in Chapter V.

3.3 Early Binding

3.3.1 Definition and Objectives

In message passing systems, early binding is used for reducing the effective overhead associated with message transfers. Overhead is the period of time during which the main CPU performs communication activities related to message preparation, communication activation, synchronization, completion notification, and possibly interrupt handling. During this time, the processor cannot perform useful computation. The goal of early binding is to reduce the effective overhead by isolating a stage in message preparation and activation, which can be performed only once and then reused

multiple times in subsequent message transfers. Early binding is a software mechanism that takes advantage of temporal locality of transfers. Two transfers are considered local temporally if they are close to each other in time or they are repeated in a loop and their message signatures are the same (*e.g.*, the user buffer, the amount of data, and the peer process are the same). The only difference between the two messages is the actual content of the user buffers. An example of communication code that uses early binding is shown in Figure 3.2.

```
init_send(IN <msg_spec>, IN <dest_spec>, OUT request) // setup
for(i = 0; i < num_iterations; i++)
      <prepare data in message buffer>          // computation
      start(IN request)                         // initiation
      wait(IN request, OUT status)              // completion
endfor
request_free(INOUT request)                     // release
```

Figure 3.2 Early binding pseudo code for sending process

This example is written in a generic communication pseudo code. If MPI is used for implementing this code segment, then the *MPI_Send_init*, *MPI_Start*, *MPI_Wait*, and *MPI_Request_free* calls that form the persistent MPI API will be used instead. The aggregate argument <msg_spec> will expand to three function parameters {buffer address, count, datatype} and the <dest_spec> argument will also expand to {destination rank, message tag, communicator}. In this code segment, message preparation, request creation, and partial request activation is performed outside of the communication loop. The entire overhead associated with these operations is incurred only once. Inside the loop, the sending process prepares the data in the message buffer and initiates the actual message transfer. This transfer includes only the communication activities that are

associated with the actual initiation of the message passing procedure, which was not performed in the request setup phase. The synchronization operation at the end of the loop ensures that the message buffer can be safely reused in the next iteration. Similarly, the receiver will perform the sequence of operations presented in Figure 3.3.

```
init_recv(IN <msg_spec>, IN <dest_spec>, OUT request) // setup
for(i = 0; i < num_iterations; i++)
      start(IN request)                          // initiation
      wait(IN request, OUT status)               // completion
      <use data in receive message buffer>       // computation
endfor
request_free(INOUT request)                      // release
```

Figure 3.3 Early binding pseudo code for receiving process

In this pseudo code, the receiver first performs early binding by creating a request object associated with the expected message. Then, inside the loop, the user thread starts and waits on the receive request multiple times. After the synchronization point indicated by the *wait* operation, the receive buffer contains the data sent by the sender, and the receiver can proceed with local computation until the next iteration.

3.3.2 Target Systems and Algorithms

This section identifies the conditions under which early binding can be used to achieve a significant improvement of communication performance through reducing the effective overhead. Using BOUM, the systems that can be identified as best candidates for utilizing early binding have communication time T_c of messages with size of m bytes $T_c(m) = o + bm$, such that $R \equiv o/bm > E$, where E is a threshold, specific to the performance optimization goals. The ratio R represents the relative importance of the

overhead o in the overall communication time T_c. This ratio has a broader scope than the widely used $n_{1/2}$ metric, which specifies the message size for which $o = bm$.

If the algorithm's communication time dominates T_p, (*i.e.*, the application is communication bound) and E is sufficiently large, than according to Amdahl's law, minimizing the overhead may lead to a significant effect on the overall performance. This effect is expressed as:

$$S = \frac{1}{1-\alpha+\alpha\frac{T_c^{fast}}{T_c^{slow}}} = \frac{1}{1-\alpha+\alpha\frac{1}{S_c}}, \tag{3.6}$$

where S is the effect of the overhead reduction on the execution time T_p (*i.e.*, the application speedup), α is the ratio between the communication time T_c and the total parallel execution time T_p (*i.e.*, $\alpha = T_c/T_p$), T_c^{fast} is the communication time after the overhead improvement, and T_c^{slow} is the communication time before the improvement. The ratio T_c^{fast} / T_c^{slow} is replaced with $1/S_c$ in the second part of the equation, which is used for defining the communication speedup $S_c = T_c^{slow} / T_c^{fast}$. The coefficient α represents the relative weight of T_c in T_p before the improvement. Furthermore, communication speedup can be expressed through R, which results in the following expression for the application speedup:

$$S = \frac{1}{1-\alpha+\alpha\frac{1+kR}{1+R}}, \tag{3.7}$$

where k is a coefficient representing the reduction of the communication overhead, $k = o^{fast}/o^{slow}$. Expression (3.7) provides a relationship between the overall application

speedup after the optimization and the relative weight of overhead in the total communication time. The practical importance of this expression is that it can estimate quantitatively the improvement of overall execution time when the overhead is reduced by a factor of $1/k$ and the relative weight of the overhead in the communication time is given as R. By using (3.7), an application developer can obtain a quick estimation of the potential benefit of early binding.

The practical interpretation of expression (3.7) is illustrated with the following example. If the ratio α between communication and overall execution time is $\alpha = T_c/T_p = 0.5$, R is 0.6, and the overhead is reduced by a factor of two ($1/k = 2$, hence $k = 0.5$), then the overall execution time of the parallel algorithm will be reduced by 10.5%. It is evident that the larger the values of α and R are, the larger the effect of overhead reduction on the overall performance becomes. On the other hand, if an algorithm spends most of its communication time in long message transfers, then the ratio $R = o/bm$ will approach zero as m grows[2]. Consequently, the factor $(1 + kR)/(1 + R)$ in (3.5) will approach one, which makes the impact of the overhead optimizations on the overall performance negligible. The threshold E can be used to determine whether the optimizations of overhead have practical significance for a given application. The value of the threshold E can be selected such that if $R < E$, the cost of achieving speedup S as a result of reducing the overhead with $1/k$ times is too high with respect to the performance benefit. Again, the value of E is user-specific and can be customized according to cost

[2] This is true, assuming that the overhead is constant with respect to the message size m.

and performance objectives. For instance, one organization may accept $E = 0.2$, which would mean that for a specific application executed on a given target platform with $R = o/bm < E$, the performance gain of early binding will be insignificant. For another organization, the same value of $E = 0.2$ may still be acceptable since it may be interested primarily in maximizing performance, not in optimizing cost. Here, cost expresses the investment for re-coding existing applications that do not take advantage of early binding, training the personnel to use early binding API's, and increased complexity of the code.

Earlier it was assumed that the message overhead o is constant and does not depend on the message size m. However, on certain systems, the overhead is dependent on the message size and can be expressed as a function of m: $o = o(m)$. Then, the transmission time will become $T_c = o(m) + bm$ and $R = o(m)/bm$. A representative of a system that exhibits such overhead is a cluster of workstations using a VIA network. VIA requires that each communication buffer be pinned in physical memory before this buffer is used in a send or receive operation. Pinning a set of physical pages is performed through a system call. The results of an experiment that measures the time for pinning and unpinning varying number of pages on Linux and Windows operating systems with Giganet VI Kernel Agent are shown in Figure 3.4. In this figure, the size of the pinned/unpinned buffers is represented as a number of physical pages. The size of the pages of both Linux and Windows is 4,096 bytes. Both runs were executed on the same hardware configuration (*sag* cluster, as described in Chapter IV and further in Appendix B).

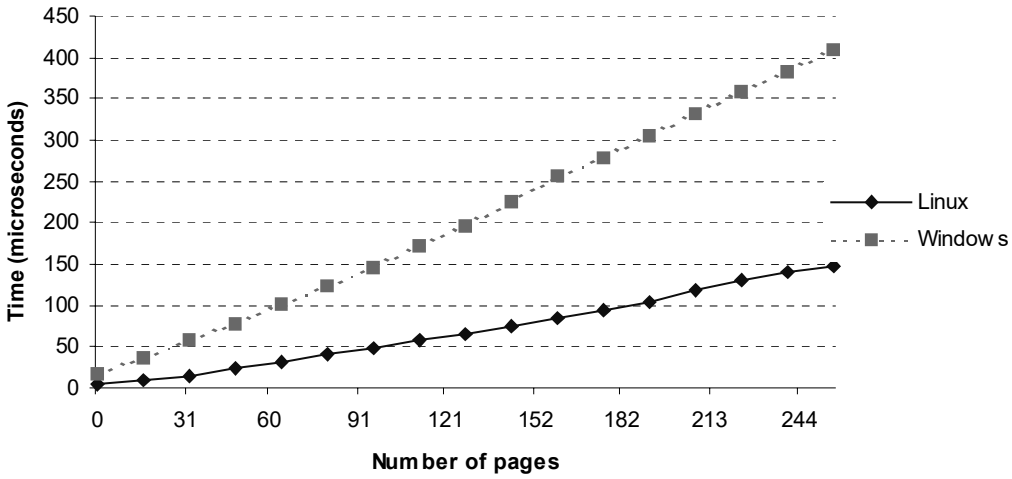


Figure 3.4 Cost of pinning and unpinning physical pages with Giganet

The time for pinning and unpinning can be represented as a summation of two components. The first component T_{sw} is constant and represents the context switches between the user process and the operating system kernel. The second component is the time T_{pin} for the actual pinning/unpinning procedure. Then, the total overhead o can be expressed as

$$o(m) = T_{const} + T_{sw} + T_{pin} \,, \tag{3.8}$$

where, T_{const} is a constant overhead in the message-passing middleware not associated with pinning and unpinning. Since T_{sw} is also constant, it can be accumulated in T_{const}, such that $o_{\text{const}} = T_{const} + T_{sw}$. Experimental results show that the time for pinning pages is linearly dependent on the number of pages m, as shown in Figure 3.4. Then, the pinning/unpinning time can be expressed as $T_{pin}(m) = vm$, where v is a coefficient that primarily depends on the operating system and the speed of the host processor. The result

for T_{pin} can be substituted in formula (3.8) for the overhead. Then the overhead of a message with size m can be expressed as:

$$o(m) = o_{\text{const}} + vm, \tag{3.9}$$

which is a linear function of m. Consequently, the ratio R becomes:

$$R = \frac{o(m)}{bm} = \frac{o_{const} + vm}{bm}. \tag{3.10}$$

In expression (3.10), both the numerator and denominator are linear functions of the message size m. As opposed to the case when the overhead o is constant and R approaches zero as the message size m grows, in this case, R will not approach zero and will be a decreasing function with a slope dependent on the relationship between the coefficients v and b, assuming $b > v$. As a result, R will become smaller than the threshold E for significantly larger message sizes than when the overhead is constant. The practical implication of (3.10) is that reducing message overhead can be an adequate performance enhancing technique in a broader class of parallel algorithms and platforms than initially assumed, after possibly measuring only the zero-length message overhead and extrapolating it to any message size $m > 0$. Special attention should be paid to systems such as clusters interconnected with Myrinet or VIA networks, which require pinning of user buffers before communication takes place. A message-passing system that strives to achieve maximum performance should account for this feature of high-speed networks and attempt to provide optimizations that minimize the impact of pinning. Early binding is one of the most natural approaches for addressing the overhead associated with pinning. Through early binding, a buffer can be pinned once and reused multiple times in

subsequent communication transactions, followed by a single unpinning operation. In fact, unpinning can be subjected to further optimizations by postponing the act of unpinning and expecting that the particular buffer can be reused at a later stage, thereby avoiding the pinning associated with a subsequent transaction. Thus, pinning and unpinning operations can be effectively removed from the overhead component associated with memory management of high-speed network software interfaces.

3.3.3 Theoretical Description

The goal of introducing early binding is to minimize the effective message transfer overhead. This is achieved by representing the overhead as a sum of two components – message setup/release overhead (o_s) and message initiation/completion overhead (o_i), such that $o = o_s + o_i$. The overhead associated with setup and release will be collectively referred to as "setup" overhead, whereas the overhead associated with initiation and completion will be denoted as "initiation" overhead. If the parallel algorithm suggests temporal locality of transfers so that the same message buffer can be reused multiple times, then early binding can effectively eliminate the setup overhead o_s from the communication time and reduce the effective overhead o to the levels of the initiation overhead o_i. If the setup overhead o_s represents a substantial portion of the total overhead o, then the overhead reduction achieved through early binding may significantly improve the communication efficiency of the algorithm. The overall performance impact of the overhead reduction depends on the ratio $R \equiv o(m)/bm$ and the coefficient $\alpha = T_c/T_p$ as indicated in expression (3.7).

Incorporating the message setup and initiation overheads, the communication time for a message with size m can be expressed as $T_c(m) = o + bm = o_s + o_i + bm$. If the buffer where this message is located is used h times, then the total communication time for these h transfers without early binding will be $T_c(h,m) = h(o_s + o_i + bm) = ho_s + ho_i + hbm$. If early binding is used, the factor ho_s can be replaced just with o_s, since the message setup phase will be performed only once and then reused $(h - 1)$ times. Then, T_c with early binding will become $T_{cEB}(h,m) = o_s + ho_i + hbm$. If $o_s \approx o_i$, then the aggregate overhead with and without early binding can be approximated respectively as $T_c(h,m) = 2ho_i + hbm$ and $T_{cEB}(h,m) = (1 + h)o_i + hbm$. For large values of h, the factor $1 + h$ can be replaced with h without significant loss of accuracy[3], so $T_{cEB}(h,m) \approx ho_i + hbm$. Using the ratio $R \equiv o/bm \approx 2o_i/bm$, we can substitute bm by $2o_i/R$ in the communication time and find that $T_c(h,m) \approx 2ho_i + 2ho_i/R = 2ho_i(1 + 1/R)$ and $T_{cEB}(h,m) \approx ho_i + 2ho_i/R = 2ho_i(1/2 + 1/R)$. In this case, the communication speedup S_c can be expressed as follows:

$$S_c \approx \frac{T_c(h,m)}{T_{cEB}(h,m)} = \frac{2(R+1)}{R+2} \tag{3.11}$$

If $R = 0.6$, as in the earlier example, then the speedup of the communication time will be 1.23. The interpretation of this result means that using early binding reduces the total communication time of transmitting h messages each with size m by 23% in respect to transmitting these h messages without early binding. Expression (3.11) allows for estimation of the practical effect of early binding on the communication performance depending on the ratio R when $o_i \approx o_s$. For instance, formula (3.11) shows that if $R > 1$,

---

[3] This assumption means that the derivations for the impact of early binding on performance represent upper bounds.

the communication speedup will be $S_c > 1.33$. R usually has higher values for short messages for which the total communication time is dominated by the overhead.

Systems using pinning of physical pages, such as VIA, have message setup overhead o_s that is substantially larger than the message initiation overhead o_i. We can represent this relationship generically as $o_s = qo_i$, where q is a factor representing how much larger (or smaller) the message setup time is with respect to the message initiation time. Then, the expression for the communication time speedup in (3.11) can be generalized as:

$$S_c = \frac{(1+q)(1+R)}{1+q+R}. \tag{3.12}$$

This generalized expression yields values greater than one for any q and R greater than zero. This means that early binding will always result in some communication improvement and, hence, overall performance improvement. Using the above example with $R = 0.6$, if $q = 5$ the communication improvement from early binding will be $S_c = 1.46$. In this case, using (3.6) and (3.12) with $\alpha = 0.5$ and assuming that the entire communication time of the algorithm is dominated by the exchange of messages with size m subjected to early binding optimization, then the total performance improvement (application speedup) will be:

$$S = \frac{1}{1-\alpha+\alpha\frac{1}{S_c}} = 1.19. \tag{3.13}$$

The same calculation with $q = 1$ (meaning that $o_s = o_i$) yields a speedup of 10%. This clearly shows that if a communication intensive algorithm ($\alpha \approx 0.5$ or more) uses

repetitive messages with a certain size m such that the ratio R has relatively high values (0.5 or higher) and the message setup overhead is equal or greater than the message initiation overhead, utilizing early binding can lead to a significant improvement of the overall application performance.

It can be concluded that early binding is an important source of performance enhancement, especially on message-passing systems that exhibit relatively high overhead with respect to the wire transmission time. An important advantage of early binding is that it is "performance transparent," as indicated in (3.12) since q and R are positive in all practical systems. The importance of performance transparency is that if the message-passing middleware provides optimizations for early binding but the user application is not designed to take advantage of these optimizations, the application will not incur any performance losses. This characteristic is important because a large number of existing parallel codes do not use early binding and slowing down these applications on a system optimized for early binding is undesirable.

3.3.4 Degree of Persistence

This subsection introduces a new metric, called "degree of persistence," denoted by d_p, which serves as a measure of the capacity of a communication system to provide early binding optimizations. The metric is defined as the ratio between the message setup overhead o_s and the entire message overhead o, including the setup and the initiation overhead:

$$d_p \equiv \frac{o_s}{o} = \frac{o_s}{o_s + o_i}. \tag{3.14}$$

The possible values of d_p are in the range [0, 1]; $d_p = 1$ for $o_i = 0$ and $d_p = 0$ for $o_s = 0$. Higher values, approaching one, indicate that the system provides opportunities for taking advantage of early binding, while small values, approaching zero, indicate that an algorithm that exploits early binding will not experience significant performance improvements. Using the defining expression for the degree of persistence (3.14) and the substitution $o_s = qo_i$ made earlier for deriving the expression for communication speedup (3.12), the degree of persistence can be expressed as follows:

$$d_p = \frac{q}{1+q}, \tag{3.15}$$

Inversely, q can be expressed as a function of d_p and then substituted in (3.12). Consequently, the expression for the communication speedup as a function of the early binding optimization can be rewritten as:

$$S_c = \frac{1+R}{1+R(1-d_p)}. \tag{3.16}$$

This expression shows that for a given ratio $R = o(m)/bm$, where $T_c(m) = o(m) + bm$, the improvement of communication time S_c resulting from early binding optimizations depends only on the degree of persistence d_p, which could be obtained following a measurement procedure as suggested in (3.14). If the degree of persistence $d_p = 0$, then clearly, $S_c = (1 + R)/(1 + R) = 1$ and hence, there is no communication speedup. Alternatively, if $d_p = 1$, the communication speedup has its maximum value at $S_c = 1 + R$, which depends only on the ratio R. The other analysis that can be made from expression (3.16) is that for a given degree of persistence d_p, the communication speedup depends only on the values of the ratio R. The two boundary cases for the values of R are $R = 0$ for

communication with no overhead and $R \rightarrow \infty$ for the bandwidth component of the communication time $bw = 0$ (*i.e.*, a message with zero-length is transmitted). For the first boundary value of $R = 0$, we find that $S_c = 1$ (*i.e.*, there is no communication speedup). For the second boundary value of $R >> 1$ we find that $1 + R \approx R$ and, therefore, $S_c = 1/ (1 - d_p)$. For small values of d_p, $S_c \approx 1 + d_p$. Consequently, from the analysis based on (3.16), it can be concluded that the communication speedup will be significant when the relative weight of the overhead o in the communication time T_c has high values (*e.g.*, $R > 1$) and when the degree of persistence has values close to its maximum ($d_p \approx 1$).

Expression (3.16) possesses strong analytical power and captures the complex interactions between user application software on one hand and the message-passing middleware and low-level communication software components of the parallel system on the other. These interactions determine the effective impact of early binding on communication performance. The analysis of expression (3.16) is based only on two simple measurements for obtaining the values of R and d_p for a given message size on the target platform[4]. Then, the parallel algorithm designer can substitute these values in the analytical expression for communication speedup and estimate the impact of the improvement from early binding. This estimation, together with expression (3.5) can be used for evaluating the trade-off between the "external" cost of incorporating early binding optimizations in the application code and the estimated overall performance benefit. The outlined sequence of steps can be used in a formal procedure for design and

---

[4] The values of d_p as a function of m could also be provided by the vendor of the parallel system.

implementation of parallel algorithms that are aware of early binding. Also, system designers and MPI developers can use these steps as a guideline for evaluating systems under development for the efficiency of their support for early binding.

The degree of persistence measures the effective capability of a parallel system to provide performance gain when an application utilizes early binding. This metric in combination with the analysis suggested by expression (3.16) is an important instrument for performance analysis of the impact of early binding on the communication and overall application performance. To the best of the author's knowledge, the degree of persistence metric and the formal analysis of early binding proposed in this section are new results in parallel processing analysis.

3.3.5 Practical Use of Early Binding

In order to demonstrate the practical use of early binding and the performance analysis introduced in this section, two parallel algorithms implemented with MPI are presented. MPI has been designed to facilitate the use of early binding through its persistent API (Message Passing Interface Forum 1994). The persistent API enables parallel programmers to write algorithms that can exploit early binding enabled by the message-passing middleware and the system communication software. Unfortunately, few MPI implementations provide sufficient optimizations for early binding, which has discouraged parallel application programmers from using the persistent MPI API[5].

---

[5] MPICH is an example of an MPI implementation that provides sub-optimal persistent communication interface.

Consequently, algorithms that are suitable candidates for early binding have often been implemented without the use of persistency.

Two example parallel algorithms are studied here to illustrate the effect of early binding on the communication and overall application performance. The experimental results of the MPI implementations of these algorithms are presented later in Chapter V. These results are obtained using MPI/Pro, an MPI implementation developed by the author as a part of this dissertation work[6]. Typical candidates for early binding optimizations are iterative solvers of systems of linear equations. The two algorithms presented in this section are from this group of applications: the Jacobi stationary solver (Burden and Faires 1985) and the Conjugate Gradient (CG) non-stationary solver (Dongarra et al 2001). Both algorithms solve the linear system $Ax = b$, where A is an n x n matrix A and b is the solution vector of size n. The input data set for both algorithms is the matrix A, the vector b, and an initial solution $x^{(0)}$ of x.

Each iteration of the Jacobi algorithm performs one matrix-vector multiplication with asymptotic complexity $\Theta(n^2)$, one vector-constant addition $\Theta(n)$, one vector-constant multiplication $\Theta(n)$, one vector assignment $\Theta(kn)$, where k << 1 is a coefficient representing the relative cost of a vector element assignment with respect to an addition operation, and a convergence check $\Theta(2n)$. Clearly, for large n, the overall algorithm complexity is dominated by the complexity of the matrix-vector multiply operation; hence, the execution time of the sequential Jacobi algorithm is as follows:

---

[6] This MPI implementation is reviewed in detail in Chapter IV.

$$T_s = F(W)t_c = n^2 t_c, \tag{3.17}$$

where t_c is the time for performing a basic compute operation on the target processor.

The parallel implementation of the Jacobi algorithm is executed on p processes, using a 1-D data decomposition of the input data set is used. Higher dimensions of data decomposition are possible for reducing the surface-to-volume ratio of the process local data sets, but the idea of the analysis presented here is not to search for the best possible parallel implementation of the Jacobi algorithm, but rather, to demonstrate the impact of early binding on the performance of one specific reasonable implementation of this algorithm. The results obtained with 1-D decomposition can also be extended to higher-dimension decompositions.

According to the selected data decomposition on p processes, the data set of each process is $W/p = n^2/p$. Each process computes a portion of the solution vector x with size n/p. In each iteration i ($0 < i \leq k$), the processes compute the local portion of the new value $x^{(i)}$ of the solution vector x and then exchange their portions with the other $p - 1$ processes. Thus, in each iteration of the Jacobi solver, every process spends $n^2 t_c/p$ seconds on computation and sends and receives $p - 1$ messages of size $m = L_e n/p$, where L_e is the size of a data element in bytes (*e.g.*, $L_e = 4$ for floats). Thus, the total parallel time for the Jacobi solver using 1-D decomposition on p processes, can be expressed with the BOUM model as follows:

$$T_p = k\frac{n^2}{p}t_c + 2k(p-1)(o+bm), \tag{3.18}$$

where, n is the size of the solution vector and k is the number of iterations for convergence of the Jacobi solver. The total communication time in expression (3.18) is presented in (3.19):

$$T_c = 2k(p-1)(o+bm). \tag{3.19}$$

The pseudo code of the proposed parallel implementation is presented in Figure 3.5. This pseudo code assumes that the input coefficient matrix A and solution vector b are already distributed among all p processes and that each process has the initial value $x^{(0)}$ of the solution vector x.

```
p := get_process_count()
r := get_local_rank()
n := get_problem_size()
convergence_condition := false
repeat
      local_x := compute_x(n/p)
      for(i = 0; i < p and i != r; i++)
            send(local_x, i)
            recv(remote_x, i)
      endfor
      convergence_condition := check_for_convergence()
while(convergence_condition == false)
```

Figure 3.5 Pseudo code for the parallel Jacobi solver

This pseudo code emphasizes the communication part of the algorithm. The actual computation (described earlier) is aggregated in a single procedure, called *compute_x*, which accepts the size of the vector as an input argument.

It should be noted that expression (3.18) and the pseudo code in Figure 3.5 assume that every process participates directly in the $p - 1$ send and $p - 1$ receive operations. However, in practical systems, the algorithm can use MPI collective

operations, such as *MPI_Gather*, *MPI_Bcast*, or *MPI_Reduce*, which could offer optimizations for reducing the communication time, thus increasing the efficiency of the algorithm. Often, such optimizations are based on the use of binary or minimum spanning trees. Then, in order to take advantage of early binding, algorithms that use collective operations should employ techniques similar to those proposed by Skjellum (1998) in his work on FastMPI. The author of this dissertation has implemented a library that follows the concepts of FastMPI. This library contains the subset of the MPI collective operations that was used in the iterative algorithms presented here. The library enabled an implementation of the Jacobi and CG solvers with the native MPI library collective communication algorithms, which, in the case of MPI/Pro, are optimized with binary trees. The interface of this library is presented in Appendix A.

For the theoretical derivation of the parallel execution time of the Jacobi (and later CG) algorithm, a linear approach for implementing the communication phase has been chosen. Thus, the algorithm implementation is independent of the collective operations of the MPI library. This choice is justified by two reasons. The first reason is that the linear approach eliminates the implementation-specific variability of the collective operations in the selected MPI library, which broadens the scope and applicability of the study. For instance, one MPI library might use collective operation optimizations while another library might use linear implementations, or optimizations different from the first library. The second reason is that the implementation of the Jacobi solver with an MPI library that offers collective operation optimizations will experience certain improvement in communication, which will often result in reducing the number of

message transfers. This will potentially reduce the effective benefit of early binding; earlier, it was shown that this benefit depends on the number of transfers over which the setup overhead can be amortized. Hence, the selected approach for deriving the theoretical expression of the Jacobi solver will give an upper bound on the improvements that result from early binding optimizations. Analyzing the upper bound of the improvement will give important insights on the capabilities of a parallel algorithm to seek performance optimizations by employing early binding.

```
p := get_process_count()
r := get_local_rank()
n := get_problem_size()
convergence_condition := false
for(i = 0; i < p and i != r; i++)
      init_send(local_x, i, sreq[i])
      init_recv(remote_x, i, rreq[i])
endfor
repeat
      local_x := compute_x(n/p)
      for(i = 0; i < p and i != r; i++)
            start(sreq[i])
            start(rreq[i])
            wait(sreq[I], status)
            wait(rreq[I], status)
      endfor
      convergence_condition := check_for_convergence()
while(convergence_condition == false)
for(i = 0; i < p and i != r; i++)
      request_free(sreq[i])
      request_free(rreq[i])
endfor
```

Figure 3.6 Pseudo code for the parallel Jacobi solver with early binding

The $2(p - 1)$ communication requests within each iteration of the proposed parallel implementation of the Jacobi solver have different message signatures, since the first $p - 1$ requests are used for sending the local portion of the vector to each of the

remaining processes and the other group of $p - 1$ requests is used for receiving the local portions from these processes. Therefore, according to the early binding requirements, the $2(p - 1)$ requests per iteration cannot be subjected to optimization with early binding. However, since the Jacobi method is iterative, the same communication transactions, with a difference only in the buffer content, can be repeated across the k iterations necessary for convergence. Thus, early binding can be applied to the transactions with the same peer process in different iterations. The pseudo code presented in Figure 3.6 demonstrates the implementation of the Jacobi solver with early binding. Using the $o = o_s + o_i$ representation of the message overhead, and substituting $M \equiv 2k(p - 1)$, the communication time from (3.19) can be expressed as follows:

$$T_c = Mo + Mbm = Mo_s + Mo_i + Mbm\,. \tag{3.20}$$

By applying early binding to the algorithm, (*i.e.*, substituting Mo_s with o_s) the communication time from (3.20) becomes:

$$T_{cEB} = o_s + Mo_i + Mbm\,, \tag{3.21}$$

which leads to an absolute reduction of the communication time of $T_c - T_{cEB} = (M - 1)o_s = (2k(p - 1) - 1)o_s$. Then, the communication speedup can be expressed as follows:

$$S_c = \frac{T_c}{T_{cEB}} = \frac{M(o_s + o_i + bm)}{o_s + M(o_i + bm)}\,, \tag{3.22}$$

where $m = L_e n/p$ is the size of the exchanged messages in bytes. Using the substitutions for the degree of persistence $d_p \equiv o_s/o$ and the ratio $R \equiv o/bm$, we find the final expression for communication speedup:

$$S_c = \frac{1+R}{1+R(1-d_p)\dfrac{M+1}{M}}. \tag{3.23}$$

For large values of $M >> 1$, the factor $(M + 1)/M$ will approach 1, which will produce the same result as expression (3.16). The analysis of the Jacobi iterative solver shows that if an algorithm can establish a communication pattern that suggests temporal locality of transfers, the communication improvement of this algorithm can be found by the generalized expression (3.16) without performing a more detailed analysis. For obtaining an upper bound of the communication improvement, and, hence, the overall algorithm speedup, the algorithm's designer need only measure the degree of persistence d_p and the ratio R for the message sizes that are subjected to early binding optimization. The results for these two metrics can then be substituted in expression (3.16), which in turn can be substituted in expression (3.6), to obtain the overall performance gain from early binding.

The second algorithm presented for illustration of the early binding effects on performance is the CG (Dongarra et al. 2001). The CG algorithm is a non-stationary, iterative solver for systems of linear equations, which uses search direction techniques for faster convergence than the Jacobi method. Similarly to the Jacobi solver, the input data set the CG is an n x n matrix A, a solution vector b of size n, and an initial value of the solution vector x. In each iteration, the CG algorithm performs one matrix-vector multiply with an asymptotic complexity $\Theta(n^2)$, three vector additions $\Theta(n)$, two vector dot products $\Theta(2n)$, one vector assignment $\Theta(kn)$, and a check for convergence of order $\Theta(n)$. Clearly, the algorithm computation time is dominated by the matrix-vector multiply as in

the Jacobi algorithm. Thus, serial execution time of the CG algorithms can be represented by:

$$T_s = F(W)t_c = n^2\, t_c, \tag{3.24}$$

which is the same as for the Jacobi algorithm. Since, the asymptotic complexity analysis accounts only for the operation with the highest complexity, this analysis ignores the lower order $\Theta(n)$ components. However, these components have higher relative cost in the CG algorithm in comparison to the Jacobi algorithm. The practical impact of this observation is that although the asymptotic analysis yields similar results for both algorithms, the increased computation cost of each iteration will reduce the relative impact of communication on the overall execution time. This, in turn, will reduce the relative effect of early binding on the performance. As a result, it is expected that the experimental results from the implementation of the CG algorithm will show lower levels of overall performance improvement from early binding than does the Jacobi algorithm. This expectation is further supported by the quality of the CG algorithm to converge much faster than the Jacobi solver, which is a result of the search direction technique employed by the CG (Dongarra et al. 2001).

The structure of the parallel implementation of the CG algorithm resembles the structure of the Jacobi algorithm. Again, a 1-D decomposition of the input data set is used. The communication pattern is similar to the Jacobi algorithm with one exception: the result of the matrix-vector multiplication is a new vector that is later used for determining the optimal direction for convergence and then for computing the actual values of the next approximation of the vector x. Again, the performance analysis of the

CG algorithm, as a result of early binding, can be performed by the generalized analysis based on expressions (3.6) and (3.16).

This section presented a theoretical description of early binding using the BOUM model and introduced a new metric, degree of persistence, for describing the capacity of a parallel system to deliver effective performance gains when applications use early binding. A generalized analysis for studying the impact of early binding on communication and overall performance was presented as well as a practical procedure for estimating this impact. Finally, this section presented two parallel iterative algorithms for solving systems if linear equations. These algorithms were subjected to early binding optimization and were analyzed with the suggested performance analysis.

3.4 Overlapping of Communication and Computation

This section presents a definition of overlapping of communication and computation and states the performance objectives of overlapping. The section focuses on the underlying hardware and system software features as well as on the applications software characteristic that are required for achieving effective overlapping. A theoretical framework that accounts for overlapping in parallel performance models is presented. Also, a new metric, called “degree of overlapping,” is introduced. Finally, a parallel algorithm implemented with overlapping is studied and its performance expressed through the presented theoretical framework. In Chapter V, a practical implementation of this algorithm is compared to a parallel implementation that does not utilize overlapping, in order to demonstrate the effective performance gain of overlapping.

3.4.1 Definition and Objectives

Overlapping of is a high-performance software mechanism that enables concurrent execution of independent communication and computation activities. The major objective of overlapping is to reduce the overall application time by utilizing hardware and software architectures that offer concurrent progress of communication and computation. Overlapping is one of the fundamental algorithmic approaches for improving parallel performance. All parallel applications could benefit from overlapping to a certain degree.

If the total execution time of a parallel application is T_p, the computation time is T_{comp}, and the communication time is T_c, then using (3.1), the objective of overlapping can be expressed as follows:

$$T_p = T_{comp} + T_c - T_o \,, \tag{3.25}$$

where, T_o is the time saving achieved through overlapping. Traditional approaches for parallel performance improvement emphasize minimization of T_{comp} and T_c and typically rely on increasing raw processor and network speeds. Clearly, overlapping presents an alternative approach for improving performance. Overlapping depends primarily on efficient software and hardware architectures, rather than on top speed components. Overlapping utilizes software techniques such as:

- asynchronous message-completion notification,
- low processor overhead, and
- independent message progress.

Major hardware features of the parallel system needed for effective overlapping are as follows:

- availability of excess bandwidth of the memory subsystem in order to sustain concurrent memory accesses for communication and computation, and
- intelligent network interface controllers capable of accessing host memory.

The performance objectives of overlapping are orthogonal to the objectives of the alternative approaches for minimizing T_{comp} and T_c; hence, these objectives can complement each other thereby, aggregating the benefits of multiple performance-enhancing approaches, rather than applying these approaches in a mutually exclusive manner.

3.4.2 Requirements for Overlapping

The requirements for achieving effective overlapping can be divided into two categories. The first category is related to the capability of the parallel system to offer a high degree of overlapping. This first category covers factors such as CPU overhead, resource contention, memory bandwidth, network infrastructure, and message-passing software. The second category is complimentary to the first one and relates to the capability of the application algorithms to take advantage of overlapping.

Major hardware prerequisites for achieving a high degree of overlapping are communication with low CPU overhead and availability of sufficient memory bandwidth. Low processor overhead frees the central processor from communication and allocates more cycles to useful computation. Intelligent network controllers facilitate low

processor overhead by eliminating the need for processor involvement in transferring data between user buffers and the network fabric.

In order to enable concurrent communication and computation, the memory subsystem of the host computer should offer sufficient bandwidth in order to facilitate simultaneous accesses of the central processor and the NIC DMA engines to the main memory. These accesses require the same resources – main memory and the memory bus. As a result, overlapped communication and computation can lead to resource contention. Figure 3.3 is a schematic of the concurrent memory accesses necessary for achieving overlapping. Increasing the memory bandwidth reduces contention. The factors that affect memory bandwidth are memory chip speeds, number of memory ports, width of the memory bus, bus clock rate, architecture of memory interconnect (bus versus switch), and optimizations for interleaved and chained accesses.

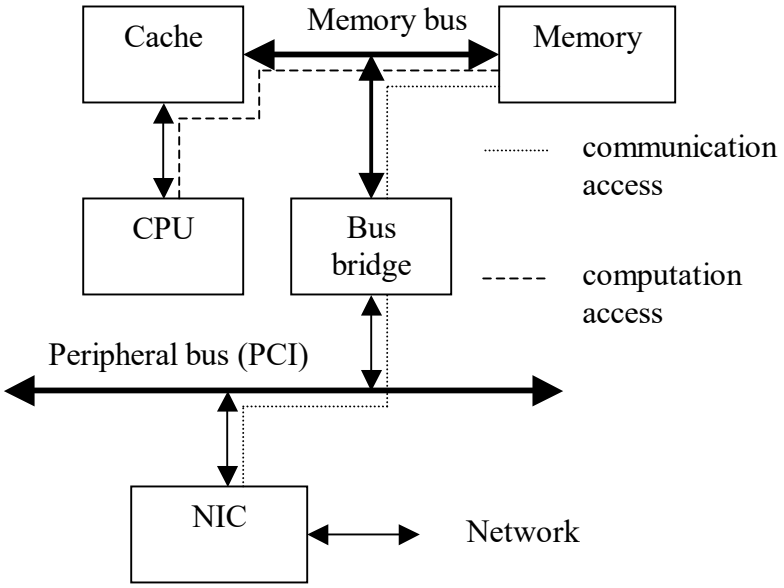


Figure 3.7. Concurrent memory accesses necessary for overlapping

The main processor cache is another factor that assists in reducing contention. The cache isolates the processor from the memory by storing data and instructions. The CPU generates main memory access requests only when there is a cache miss at the lowest cache level. The availability of large local memory on the NIC also reduces contention. Large local memory allows more data to be stored on the NIC, which reduces the number of accesses to the main memory. Fewer accesses lead to better utilization of the memory and peripheral buses with smaller overheads.

Quantification and formal representation of all factors that affect contention and memory bandwidth is a rather complex task. In this work, these factors will be represented indirectly by a newly introduced metric called "degree of overlapping," which will be determined empirically by a measurement procedure described below. Capturing the variable effects of the numerous memory contention factors in overlapped accesses is one of the major objectives for introducing the degree of overlapping.

The first category of message-passing middleware requirements for achieving a high degree of overlapping are asynchronous notification completion, independent message progress, and low CPU overhead. These issues and their impact on overlapping are discussed in detail in Chapter IV, and, for brevity, are not repeated here.

The second category of requirements for achieving effective overlapping is related to application algorithms. The designer of the algorithm should isolate independent communication and computation activities that can be overlapped and should order these activities so that the algorithm correctness is preserved. Typical algorithms that can benefit from overlapping are data-parallel, structured algorithms with

a regular communication pattern such as parallel FFT, parallel matrix transpose, and parallel sorting. Parallel algorithms that follow the data-flow paradigm (in contrast to the data-parallel paradigm) are also suitable for overlapping. Although the primary focus of this dissertation is on system-oriented methods for improving performance, guidelines for creating optimal parallel algorithms that can benefit from overlapping and early binding are also provided. An implementation of parallel FFT with overlapping is presented later in this section.

3.4.3 Theoretical Description of Overlapping

Overlapping requires that the parallel code is structured in a manner that enables two (or more) independent communication and computation activities to be executed concurrently, without violating the correct semantics of the algorithm. Programmatically, overlapping can be represented by the following pseudo code segment in Figure 3.8.

```
local_computation_1
start(IN communication_request)
local_computation_2
wait(IN communication_request, OUT status)
```

Figure 3.8 Pseudo code for overlapping of communication and computation

In this code segment, *communication_request* represents the activity associated with communication and *local_computation_2* represents the computation that is overlapped with communication. The relationship between the communication and computation activities plays an important role in achieving effective overlapping. To illustrate this importance, a computation activity X and a communication activity Y are reviewed. The durations of these activities are, respectively, t_x and t_y. For simplicity of

presentation, it is assumed that the entire computation and communication times of the application can be represented as a sum of N individual activities, such that $T_{comp} = \sum t_x$ and $T_c = \sum t_y$. This assumption leaves out initialization communication and computation procedures that are needed for preparation of the main part of the algorithm. These initialization activities are assumed to have minimal impact on the overall execution time. In the presentation below, another important assumption is made that the two reviewed activities X and Y can be executed concurrently with no interdependency. This assumption has two consequences. First, the X and Y activities should be algorithmically independent (*i.e.*, they can be executed in an arbitrary order with respect to each other without violating the correct semantics of the algorithm). Second, there must be at least two independent active processing elements that can guarantee the concurrent progress of X and Y. Such processing elements could be the central processor and an intelligent NIC.

If the reviewed communication and computation activities are not overlapped, the serial time for their completion will be $t_s = t_x + t_y$. If the two activities are ideally overlapped, the total time t_o will be the larger of t_x and t_y (*i.e.*, $t_o = \max(t_x, t_y)$). The overall time reduction as a result of overlapping is $N(t_s - t_o)$, and the effective application speedup is as follows:

$$S = \frac{Nt_s}{Nt_o} = \frac{t_x + t_y}{\max(t_x, t_y)}. \tag{3.26}$$

It should be noted that as opposed to T_{comp} and T_c, which can be represented as summations of individual computation and communication activities with times t_x and t_y, T_o *is not* a summation of the individual overlapped times t_o. T_o is a quantitative

representation of the aggregate *saving* from overlapping, while t_o is the *period of time* during which two concurrent activities are overlapped. This distinction is important for proper interpretation of the theoretical derivations and experimental results presented below.

While the theoretical description of early binding was primarily focused on the communication speedup, which was then related to the overall application speedup, the theoretical description of overlapping requires a representation of application's performance as a combination of communication and computation times simultaneously. Therefore, the analysis of overlapping does not provide explicit expressions of communication and communication speedups individually.

In order to quantify the speedup because of overlapping, the relationship between X and Y is represented as $t_y = qt_x$, where $q > 0$ is a factor that represents how much Y is longer or shorter than X. Then, the speedup is as follows:

$$S = \frac{t_x + qt_x}{\max(t_x, qt_x)} = \frac{(1+q)t_x}{\max(t_x, qt_x)}. \tag{3.27}$$

In the first case, let $q > 1$, that is, the communication time is longer than the computation time. Then, $\max(t_x, qt_x) = qt_x$ and $S = t_x(1 + q)/qt_x$. Consequently, the final expression for the speedup from (3.27) in the first case is $S = (1 + q)/q$. This expression shows that when q approaches 1 (denoted by $q \to 1$) then $S \to 2$ (a well-known upper bound), and when $q \to \infty$ then $S \to 1$. In the second case, the values of q are in the range $0 < q \leq 1$, meaning that the communication time t_y is shorter than the computation time t_x. Then, $\max(t_x, qt_x) = t_x$ and $S = 1 + q$. In this case, for $q \to 0$, $S \to 1$ while for $q \to 1$, $S \to 2$. The results of

the two cases with respect to the relationship between the durations of X and Y can be summarized as follows:

$$S = \begin{cases} \dfrac{1+q}{q}, \text{for } q > 1, t_y > t_x \\ 1+q, \text{ for } 0 < q \le 1, t_y < t_x \end{cases} \tag{3.28}$$

From (3.28), it can be seen that the speedup because of overlapping is in the range $1 \le S \le 2$, with the maximum achieved for $t_x = t_y$, while the minimum is achieved when one of the computation and communication activities is infinitely longer than the other one. This conclusion can be used as a guideline for writing parallel algorithms with overlapping. In order to take maximum advantage of overlapping, the algorithm should be divided into a sequence of computation and communication activities that are approximately of the same duration. Meeting this requirement depends on numerous platform-dependent parameters such as processor speed, network bandwidth, and system software architecture, as well as on the parallel algorithm and its data decomposition. The dependency on platform-specific factors means that the same algorithm executed on different parallel platforms can exhibit different speedups. This makes theoretical modeling of overlapping difficult and intimately dependent on the current state of the art of the technology. Furthermore, in order to take maximum advantage of overlapping, the algorithm may require different data partitioning and organization of the main computation loops on different platforms. This fact affects negatively the use of overlapping in theoretical models and its utilization in practice.

A more precise description of overlapping is necessary. Algorithm designers can use this more precise description in order to parameterize overlapping. When these algorithms are executed on different platforms, the actual values of the overlapping parameters can be adjusted accordingly. This technique is now widely used for adjusting cache-optimized algorithms on different platforms to varying cache sizes. Cache-optimized algorithms that are written in a flexible manner use parameters for the cache size and determine the actual cache size at compile or run time. Thus, these algorithms can maximize their performance gain from cache optimizations on platforms with different cache attributes. The goal of this work is to achieve similar flexibility for overlapping. Then, applications that take advantage of overlapping can be written using a set of parameters whose values are empirically determined for each target platform.

3.4.4 Degree of Overlapping

This work introduces a new metric used for accurate modeling of overlapping of communication and computation, "degree of overlapping," denoted by d_o. Degree of overlapping is used as a quantitative description of the capabilities of a system in order to perform effective overlapping of communication and computation. This quantification is necessary for devising a realistic model of the parallel execution time, which incorporates overlapping and is expressed in the form:

$$T_p = T_{comp} + T_c - T_o(d_o) \tag{3.29}$$

where the savings from overlapping T_o is a function of d_o and d_o is selected so its range of values is $0 \leq d_o \leq 1$. Furthermore, d_o is chosen so that the maximum impact of

overlapping on the overall performance is achieved as the value of d_o approaches one. There are several reasons for introducing the degree-of-overlapping metric. First, it can be used to distinguish between hardware platforms and software systems that have specific optimizations for achieving effective overlapping and those that do not provide such optimizations. Second, this new metric helps to provide more realistic model of overlapping that accurately quantifies its performance benefits to applications. Third, software and hardware designers can use the degree-of-overlapping metric as a guideline for improving parallel platform architectures and application algorithm implementations.

Earlier in this section, the application speedup resulting from overlapping was derived in (3.27) and (3.28). These derivations assumed that the overlapped activities X and Y were ideally concurrent. However, in practice there may be a significant internal interdependency between these two activities. This interdependency is primarily caused by two factors: the CPU overhead for communication and the contention for system resources. The communication activities consist of two distinct phases – setup/release and message transmission. The first phase involves message preparation, intermediate memory allocations or copies, request submission to the messaging layers, interaction with the network hardware for message initiation, completion notification, and finally release of system resources allocated for this request. All of these activities require participation of the CPU. This participation is referred to as CPU overhead – the period of time during which the processor is involved in communication-related activities and cannot be used for other computations (*e.g.*, application).

The second factor that limits the degree of overlapping is contention for system resources, most notably, the memory subsystem throughput. In order to enable effective overlapping, the memory subsystem should be capable of delivering sufficient bandwidth, so that both communication and computation can be executed simultaneously, without contention. In real systems, this is rarely achievable. Even on systems that possess high memory bus bandwidth, certain overhead is necessary for bus arbitration or serializing the access to the shared memory resource. Processor caches help minimize the memory contention by reducing the effective cost of processor accesses to memory.

In summary, ideal overlapping is impossible because the communication activities require certain processor overhead and the message transfers compete to some degree with the central processor for access to the main memory. Furthermore, the architectures of the system communication software and the message-passing middleware have a significant impact on the effective overlapping delivered to the application[7]. The degree-of-overlapping metric d_o is specifically introduced to capture these dependencies and to assist with a precise incorporation of overlapping in models for parallel programming. A degree of overlapping with value $d_o = 1$ means that the system can achieve ideal overlapping, while a value $d_o = 0$ means that overlapping will have no effect on performance. The latter case is equivalent to executing the two "concurrent" communication and computation activities in sequence, although the structure of the algorithm may suggest that these activities should be executed in parallel.

---

[7] The impact of the message-passing middleware on overlapping is studied in Chapter IV.

As a result of the CPU overhead analysis, the example communication activity Y, introduced earlier, can be represented as a sum of two sub-activities Y_h and Y_c, where Y_h denotes all activities of Y that cause processor overhead and resource contention, and Y_c is the component that can be overlapped with computation. During Y_h, no overlapping is possible because the CPU is busy with communication-related work. Therefore, the computation activity X can be overlapped only with Y_c. It is important to note that Y_h and Y_c are not two distinct phases of Y. For instance, such phase-based division could lead to Y_s and Y_t phases, where Y_s is the setup phase and Y_t is the transmission phase. However, since processor overhead and resource contention can be attributed to both Y_s and Y_t, this division does not isolate the activities that affect the degree of overlapping. Therefore, for the purposes of this work, the logical division to Y_h and Y_c is selected, rather than the temporal division. Following this logical division, the time of activity Y can be represented as a sum of two components: $t_y = t_{yh} + t_{yc}$. Then, the serial time of X and Y is $t_s = t_x + t_{yh} + t_{yc}$ and the overlapped time is:

$$t_o = t_{yh} + \max(t_x, t_{yc}) = \begin{cases} t_{yh} + t_x, & \text{for } t_x > t_{yc} \\ t_{yh} + t_{yc} = t_y, & \text{for } t_x \le t_{yc} \end{cases}. \tag{3.30}$$

Clearly, the longer the overhead time t_{yh} is, the smaller the effect of overlapping will be, since during this time no overlapping can be achieved. Alternatively, if the component Y_c is dominant, then the effect of overlapping will be higher. These relations can be used for a formal definition of the degree of overlapping through the overhead and overlapped communication times:

$$d_o \equiv \frac{t_{yc}}{t_y} = \frac{t_{yc}}{t_{yh} + t_{yc}}. \tag{3.31}$$

The boundary values of d_o can be found by using expression (3.31). First, when $t_{yh} = 0$, $t_{yc} = t_y$ and $d_o = 1$. Alternatively, when $t_{yc} = 0$, $t_{yh} = t_y$ and $d_o = 0$. The cases with $d_o \approx 0$ can be attributed to communication systems that do not have independent processing facilities for communication and rely entirely on the central processor for data transmission, or systems that have a high level of resource contention. Using (3.31), we can find that $t_{yc} = d_o t_y$ and $t_{yh} = (1 - d_o)t_y$. Then, substituting in (3.30), we find that:

$$t_o = \begin{cases} t_x + (1 - d_o)t_y, \text{ for } t_x > t_{yc} \\ t_y, \text{ for } t_x \leq t_{yc} \end{cases}. \tag{3.32}$$

Expression (3.31) plays an important role in the theoretical framework and is used later for estimating the performance gain of overlapping. Special attention is paid to the case in which $t_x > t_y$, since this condition can be easily identified by the users after a simple experiment for individually measuring the t_x and t_y times. Note that the actual precise condition is with respect to t_{yc}, not to t_y, but for practical purposes of simplicity, the more relaxed condition with respect to t_y is used.

In summary, the values of the degree of overlapping are indeed in the range $0 \leq d_o \leq 1$, which satisfies the requirement of the definition of this metric to quantify the effect of overlapping, as suggested in (3.29). At this point, it should be stated that expression (3.31) used for defining the degree of overlapping assumes that t_{yh} is constant and will not increase as the actual overlapping takes place. The opposite assumption means that the overlapped time t_o is longer than the sequential time t_s when the activities X and Y are

executed concurrently (*i.e.*, $t_o > t_s$). This condition is only possible when certain hardware or software architectural deficiencies are present in the system under investigation. An example of such a deficiency is demonstrated in Chapter V, when the capability of the message-passing middleware to support asynchronous processing is studied. Systems with such deficiencies are of little practical significance and are used only for illustration here.

An important practical consequence of the degree of overlapping definition and expression (3.32) is that when $T_{comp} \geq T_c$ and the entire execution time of the algorithm (or at least the predominant portion of it) can be represented as overlapped communication and computation activities, then the expression for the parallel time from (3.29) can be re-written in the form:

$$T_p = T_{comp} + (1 - d_o)T_c, \tag{3.33}$$

which leads to the definition of the time saving T_o of overlapping as a function of the degree of overlapping and the communication time.

$$T_o(d_o) = d_o T_c, \tag{3.34}$$

which actually is the function sought in (3.29). Again, expressions (3.33) and (3.34) are special cases and are valid only if the condition $T_{comp} \geq T_c$ is met.

The analysis of cases when this condition is not met is possible and may be a subject of future work, following the findings of this dissertation. This additional analysis may yield important practical results for applications that are communication bound (*i.e.*, the communication time dominates the total execution time). Such algorithms usually

exhibit poor scalability and overlapping can be used as an important source for improving their performance, so this future effort appears useful.

3.4.5 Procedure for Determining Degree of Overlapping

Finding the degree of overlapping d_o is based on expression (3.32). The value of d_o can be computed if t_o, t_x and t_y are known in this expression. In order to illustrate the procedure for determining the values of d_o, we again review the two activities X (computation) and Y (communication), with durations t_x and t_y, respectively. The individual times t_x and t_y can be measured by independently executing the two activities X and Y. Then, the overlapped time t_o is measured by starting the two activities together and observing the moment when the longer activity finishes. Obviously, $t_o \geq \max(t_x,t_y)$.

Using the case $t_x > t_{yc}$, from (3.32) we find that $t_o = t_x + (1 - d_o)t_y$, hence the expression for determining the values of the degree of overlapping metric with practical means is as follows:

$$d_o = 1 + \frac{t_x - t_o}{t_y} = \frac{t_s - t_o}{t_y}. \tag{3.35}$$

In this expression, all parameters are empirically measurable. In order to guarantee that $t_x > t_{yc}$, the computation activity X can be chosen so that $t_x > t_y$, which guarantees that $t_x > t_{yc}$ since $t_{yc} = t_y - t_{yh}$ and t_{yh} is non-negative.

The analysis of (3.35) shows that for $t_o = t_x$, the degree of overlapping has its maximum value $d_o = 1$ (*i.e.*, the system provides ideal overlapping). For any real case, $t_o > t_x$; hence, the value of d_o will be less than one. The lower bound of d_o is found when the component $(t_x - t_o)/t_y = -1$. This condition is met when $t_o = t_x + t_y$, in which case the

overlapped time is equal to the serial time, $t_o = t_s$ (*i.e.*, there is no effect of overlapping). The degree of overlapping, as expressed in (3.35), can become negative for the cases when $t_o > t_s$. This means that, when attempting overlapping, the application actually has negative gain of performance or a performance loss. Earlier in this section, systems with such a property were identified to have architectural deficiencies that cause the overlapped time of two activities to become longer than the sum of the sequential times of these activities.

It is important to distinguish the case when the effective performance loss ($t_o > t_s$) is caused by architectural deficiencies from the case when the applications experience diminishing or negative returns from overlapping because of algorithmic and data set specifics. Often algorithms with overlapping suggest breaking longer messages in series of small segments transmitted in a pipelined fashion and overlapped with computation. This approach is called "segmentation." As a function of the number of segments and the corresponding increase of overhead, the execution time of the overlapped communication and computation activities may become larger than the sequential times. The segmentation approach for designing algorithms with overlapping and its impact on performance are studied by Baden and Fink (1998) and Sohn et al. (1999), and are further investigated in this section.

3.4.6 Practical Use of Overlapping

This section presents an example algorithm that is implemented using overlapping of communication and computation. This algorithm is the Cooley-Tukey 1-D, radix-2

algorithm with decimation in frequency for computing an n-point FFT (Proakis and Manolakis 1996). This FFT algorithm has an asymptotic complexity of $\Theta(n\log n)$ operations. Assuming that a basic compute operation is performed for time t_c seconds, the serial time for execution of the algorithm is given by:

$$T_s = t_c n\log n, \tag{3.36}$$

where t_c is the computation time on each complex element of the FFT algorithm. On average, the computation for obtaining a new FFT value in the implemented algorithm results in two floating-point multiplies and three floating point additions.

The parallel FFT algorithm presented here partitions the linear input vector of size n elements to p processes (assuming that $n = 2^d$ and $n/p = 2^k$, $k > 1$), so that each process operates on a contiguous block of size $m_e = n/p$ elements. The binary-exchange method for communication is used.

The algorithm is divided into two phases. During the first phase of $\log p$ iterations, the vector combination involves elements residing on different processes. In the second phase of $\log n$ - $\log p$ iterations, the vector combinations are local. The vectors that are communicated between the processes in the first phase are of size m_e elements. Consequently, the total communication time for a process is given by:

$$T_c = \log p T_c(m), \tag{3.37}$$

where $m = m_e L_e$ is the size of the exchanged messages in bytes and L_e is the size of one element of the FFT computation in bytes (*e.g.*, $L_e = 8$ for complex single-precision floating-point computation). The total parallel execution time of the algorithm without applying overlapping is as follows:

$$T_p = \log p T_c(m) + \log n(t_c m_e). \quad (3.38)$$

As mentioned earlier, the FFT algorithm can be logically divided into two phases – global and local. Then, the parallel execution time can be represented as:

$$T_p = T_{global} + T_{local} = \log p(t_c m_e) + \log p T_c(m) + (\log n - \log p)(t_c m_e). \quad (3.39)$$

The local phase of the algorithm contains no communication; hence, this phase can be eliminated from the overlapping optimization. Such an optimization can be applied only to the global phase, which contains both communication and computation components. The impact of overlapping on the overall performance obviously depends on the relative weight of the global phase T_{global} in the total execution time T_p. The presentation below focuses on the global phase.

The opportunity for overlapping arises from the fact that in each step during the global phase of the FFT algorithm, all processes exchange data of size m bytes with a corresponding peer process. Before proceeding with local computation, the processes must wait until the message from the peer process arrives. For large problem size $W = \Theta(n)$, the size of the exchanged message in each step will become significant and the transmission time of this message, $T_c(m) = o + bm$, will have a sizeable impact on the time spent in the specific step.

The FFT computation does not require that the entire peer vector of $m_e = m/L_e$ elements be received before the computation begins. This feature of the FFT enables a division of the message of size m bytes into s segments of size $m_s = m/s$ bytes and a division of the computation on all m_e elements into s independent computations on sub-vectors of size m_e/s. Then, the transmission of the message segments can be pipelined and

overlapped with computation. Figure 3.9 presents the pseudo code for the suggested FFT algorithm with overlapping of communication and computation in the global phase.

```
schedule receive transfer for segment 0
schedule send transfer for segment 0
for(j = 1;  j < s; ++j)
      schedule receive transfer for segment j
      schedule send transfer for segment j
      wait for receive transfer for segment j - 1
      wait for send transfer for segment j - 1
      Compute segment j - 1
endfor
wait for send transfer for segment s - 1
wait for receive transfer for segment s - 1
compute segment j - 1
```

Figure 3.9 Pseudo code for the global phase of the FFT algorithm with overlapping

From (3.39), the execution time of the global phase T_{global} of the FFT algorithm can be represented as follows:

$$T_{global} = \log p(t_c m_e) + \log p T_c(m) = \log p(t_c m_e + T_c(m)), \tag{3.40}$$

which can, in turn, be expressed as $T_{global} = \log p T_{step}$, where the time of each step of the global phase is $T_{step} = t_c m_e + T_c(m)$. This representation of T_{step} and the nature of the FFT algorithm meet the requirements for overlapping, as stated earlier in this section. Hence, an overlapping optimization is applied to the communication and computation components in T_{step}. By incorporating the described segmentation procedure, the time of a global step can be expressed as follows:

$$T_{step} = st_c \frac{m_e}{s} + sT_c(\frac{m}{s}) . \tag{3.41}$$

Before the computation in the step can begin, each process must receive the first segment and, respectively, send one of its segments to the peer process. During the time

of these transfers, the process cannot perform computation; so, no overlapping can be achieved during the exchange of the first segment. All of the remaining $s - 1$ subsequent message segments can be overlapped with computation involving vector elements from earlier segments. Thus, during iteration i ($1 < i < s$) from a given global step, each process executes three procedures:

- P_1: waiting for completion of the exchange of segment $i - 1$,
- P_2: scheduling a send and a receive operation for exchanging segment i, and
- P_3: performing computation on the sub-vector of segment $i - 1$.

The initialization stage of the algorithm schedules the transfers of segment zero (procedure P_1) while the finalization stage is performing the synchronization (P_2) and computation (P_3) procedures on the last segment. Thus, overlapping of communication and computation is achieved between the P2 and P3 activities in $s - 1$ segments. Aggregately, one of the segments is not subjected to overlapping because of the initialization and finalization procedures. Taking this into account, the time of the step from (3.40) can be re-written as a sum of four terms:

$$T_{step} = T_c(\frac{m}{s}) + (s-1)T_c(\frac{m}{s}) + (s-1)t_c\frac{m_e}{s} + t_c\frac{m_e}{s}. \tag{3.42}$$

The first and the fourth terms represent the initialization and finalization activities that are not overlapped. The second and the third components are the communication and computation times of the $s - 1$ segments that are overlapped. Then, applying (3.33) and assuming that $t_c m_e/s \geq T_c(m/s)$ we finally find the time for the execution time of a global step:

$$T_{step} = T_c(\frac{m}{s}) + (s-1)T_c(\frac{m}{s})(1-d_o) + (s-1)t_c\frac{m_e}{s} + t_c\frac{m_e}{s}, \tag{3.42}$$

which could be further simplified into the form:

$$T_{step}(s) = t_c m_e + T_c(\frac{m}{s})(s-(s-1)d_o). \tag{3.43}$$

Expression (3.43) gives the time for each step of the global phase of the parallel FFT algorithm with overlapping as a function of the number of segments s. The total time for the global phase is obtained by multiplying the result from (3.43) by log$p$, which can in turn be substituted in the total execution time expression (3.39).

The parallel implementation of the FFT algorithm with overlapping performs best when the degree of overlapping has its maximum value, $d_o = 1$. Then, $T_{step} = t_c m_e + T_c(m_s)$, which effectively hides $s - 1$ of the total number of s transfers of the segments with size $m_s = m/s$. If the degree of overlapping has its minimum value, $d_o = 0$, then $T_{step} = t_c m_e + sT_c(m_s)$, which means that there is no effect of overlapping and the step execution time is equivalent to the time $T^{no}_{step} = t_c m_e + T_c(m)$ without applying the segmentation procedure. In fact, because the ratio $T_c(m)/(sT_c(m_s))$ is always smaller than one, the implementation with overlapping exhibits a slowdown with respect to the implementation without overlapping when $d_o = 0$. This is a result of the increased cumulative overhead of the multiple transfers needed to exchange the entire amount of data m in s individual segments.

An important element of the descriptive power of the analysis based on the degree of overlapping is that this analysis implicitly incorporates issues related to scaling the problem size n and the number of processes p. Also, of great importance is the analysis of

the optimal number of segments used for achieving effective overlapping. Evidently, under ideal circumstances, increasing the number of segments s leads to reducing the $T_c(m_s)$ component of T_{step}, as described in (3.43). However, increasing the number of segments, evidently decreases the size of the segments; hence, the relative weight of the overhead increases. This will naturally reduce the capabilities for effective overlapping, since the CPU must spend more of its time on overhead processing. Sohn et al. (1999) realize that increasing the number of segments leads to an increase of the overhead, which impairs the performance improvements from overlapping. The abstraction power of the analysis, based on the degree-of-overlapping metric, captures this behavior and enables the designers of the parallel algorithm to estimate the impact of such relationships as the increase of overhead while increasing the number of segments.

In order to facilitate the scalability aspect of the overlapping analysis, another metric is introduced, called "segmentation efficiency." This metric is similar to the "communication efficiency" metric proposed by Sohn et al. (1999), which was reviewed in detail earlier in Chapter II. As opposed to the communication efficiency, the segmentation efficiency metric is defined as:

$$E_s = \frac{T_c(m)}{sT_c(m_s)} = \frac{o(m) + bm}{so(m_s) + sbm_s}, \tag{3.44}$$

where s is the number of segments in which a message with size m is divided and $m_s = m/s$. Obviously, $sbm_s = sbm/s = bm$. However, $so(m_s)$ is not equivalent to $o(m)$, which results from the increased cumulative overhead of segmentation. The possible values of E_s are in the range (0, 1). A maximum value of one represents the case when the

cumulative overhead does not grow with increasing the number of segments. Then, perfect segmentation efficiency is observed. Inversely, a value of zero represents the case when the overhead has increased infinitely for the given number of segments. In real systems, the boundary values of the segmentation efficiency are unattainable.

The segmentation-efficiency metric can be used to evaluate the performance gain of overlapping based on segmentation with a given number of segments. As mentioned earlier, by reducing the message size of the segments when the number of segments is increased, the degree of overlapping for the shorter message will be lower. For some number of segments s, the message segment size m_s and the degree of overlapping d_o will become such that the time of the step with overlapping $T_{step} = t_c m_e + T_c(m_s)(s - (s-1)d_o)$ will become equivalent to the time of the step without overlapping (*i.e.*, without segmentation) $T^{no}_{step} = t_c m_e + T_c(m)$. Then, in order to find the condition for this event, we can solve the equation:

$$t_c m_e + T_c(m_s)(s - (s-1)d_o) = t_c m_e + T_c(m) \tag{3.45}$$

with respect to E_s. Obviously $s t_c m_s = t_c m_e$, (excluding possible cache effects); hence equation (3.45) becomes:

$$E_s = 1 - \frac{(s-1)}{s} d_o \tag{3.46}$$

Expression (3.46) can be used as a boundary condition that determines for what number of segments s the effect of overlapping becomes zero, and further increase of the number of segments will only lead to performance loss. Algorithm designers can use the analysis suggested by (3.46) in order to determine the upper bound of s so that the

performance gain from overlapping will be positive. The optimum number of segments s_b is in the range (1, s_m), where s_m is the value of s that satisfies equation (3.46). The value of s_m can be obtained by first performing a measurement of $d_o = d_o(m)$ for some range of message sizes that are present in the parallel algorithm and then building a family of empirically obtained curves $E_s = E_s(s, m)$, where m is the message size that is used for overlapping and s is a variable in the range [1, max) and max is sufficiently large. Then, by substituting specific values of d_o and E_s, the designers can obtain s_m.

Although the suggested analysis can yield important boundary values for s, the practical procedure for determining s_m can be quite tedious and may also require a large number of experiments. Below, a theoretical analysis for the overlapped execution time as a function of the number of segments is presented. This analysis expresses s_m and more importantly s_b only through the parameters R and d_o. Thus, by only two measurements (for obtaining R and d_o), the designer can estimate the optimum number of segments that will yield the best overall execution time. The first objective is obtaining an analytical expression for s_m. Substituting (3.44) in equation (3.46) yields:

$$\frac{T_c(m)}{sT_c(\frac{m}{s})} = 1 - \frac{(s-1)}{s} d_o . \tag{3.47}$$

Further, using the representation of the communication time according to BOUM $T_c(v) = o + bv$, and assuming for simplicity that the overhead is invariable on the message size, we obtain:

$$\frac{o + bm}{so + bm} = 1 - \frac{(s-1)}{s} d_o , \tag{3.48}$$

which is a quadratic equation with unknown s. This quadratic equation has two solutions as follows:

$$s_1 = 1\,,\ s_2 = \frac{bmd_o}{o(1-d_o)} = \frac{d_o}{R(1-d_o)}\,. \tag{3.49}$$

The first solution is as expected $s = 1$, which represents the case with no overlapping. The second solution presents the second value of s for which the step execution time in the global phase of the parallel FFT with overlapping is equivalent to the time of the same algorithm without overlapping.

The analysis of the optimum value s_b of the number of segments s is based on expression (3.43) for the execution time of one step as a function of s. The standard procedure for determining an extremum of a function is applied:

$$\frac{dT_{step}(s)}{ds} = 0\,. \tag{3.50}$$

The solution of this equation produces two values of s – one negative and one positive. Evidently, the negative value has no practical meaning. Hence, the only solution that can be used for the optimum analysis of the execution time with overlapping is the positive value, which is as follows:

$$s_b = \sqrt{\frac{d_o}{R(1-d_o)}}\,. \tag{3.51}$$

In summary, the overall performance (execution time) of a step of the parallel FFT with overlapping will be higher than the performance of the step without overlapping for $s \in [s_1, s_2]$ as found in (3.49). The maximum performance gain will be

obtained for $s = s_b$ where $s_1 < s_b < s_2$ and s_b is the square root of s_2, which is represented in Figure 3.10.

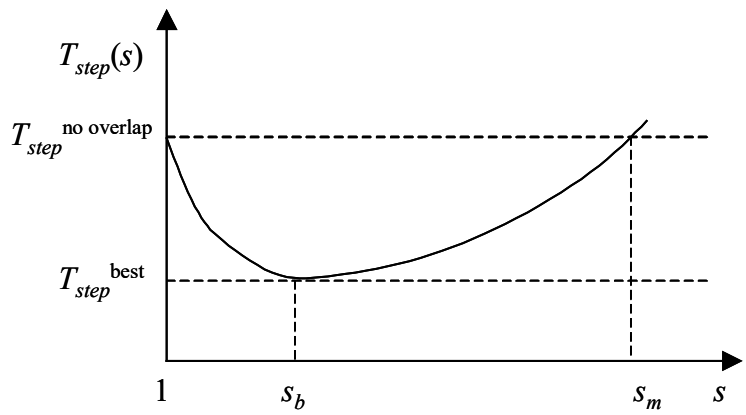


Figure 3.10 Impact of number of overlapped segments on performance

The significance of the analysis presented in this section is that the performance gain from overlapping of communication and computation can be analytically expressed through a formal process based on theoretical modeling. This formal process provides a high level of abstraction for describing the complex interactions among the components of a parallel system and at the same time suggests a procedure for practical estimation of the performance gain from overlapping of an algorithm on a specific target platform.

3.5 Additional Performance Metrics

A set of new performance metrics was introduced in this chapter in order to achieve an accurate analysis of early binding and overlapping of communication and computation. These metrics are degree of persistence, degree of overlapping, and segmentation efficiency. To further assist this analysis, this section defines two additional

metrics: degree of asynchrony and CPU overhead. These metrics reflect commonly used concepts in parallel processing and communication systems. These concepts have a specific meaning in this study – they reveal important qualities of the parallel system to support effective overlapping of communication and computation.

Although the degree of overlapping metric introduced in the previous section captures the influence of the asynchrony and CPU overhead on effective overlapping, the author considers the two additional metrics as important tools for studying the hardware and software support for overlapping in more detail. While the degree of overlapping is still the definitive metric for measuring the capability of a parallel system to enable effective overlapping, CPU overhead and degree of asynchrony can give important insight on why a particular system exhibits a certain degree of overlapping. The answer to this question may assist algorithm designers and system architects to analyze the performance behavior of a parallel system with a high level of accuracy.

Degree of asynchrony, annotated with d_a, is a metric that describes the capacity of a communication system to perform asynchronous progress of messages. This feature was identified as critical for achieving high degree of overlapping. The degree of asynchrony will be measured empirically through a specifically designed test presented in Chapter V. The possible values of d_a are in the range [0, 1]. A value of one suggests that the parallel system guarantees progress of messages even if the user application never calls the message-passing library after the submission of a communication request. A value of zero suggests that the system cannot perform independent message progress (*i.e.*, applications should call the message-passing library often in order to ensure progress).

CPU overhead is the portion of the CPU time spent on communication activities during a given period. Low CPU overhead is an important factor for achieving a high degree of overlapping. The values of the CPU-overhead metric are also in the range [0, 1]. Overhead with value of one suggests that the CPU is busy with communication activities for the entire duration of the communication request. This in turn means, that no overlapping with computation can be achieved. In contrast, zero CPU overhead suggests that the CPU effectively does not participate in communication and all of its cycles can be allocated to computation. The CPU overhead is measured by observing the load of the system while communication is taking place. The experimental results of such measurements are presented in Chapter V.

3.6 Conclusions

This chapter presented a theoretical framework for describing early binding and overlapping of communication and computation as important mechanisms for improving parallel performance. This framework enables modeling of algorithms executed on parallel systems that provide efficient support for these mechanisms. The framework imposes special requirements on the models for parallel computation. It was assessed that existing models, such as BSP and LogP, are insufficient for meeting these requirements. The BOUM parallel model was developed to incorporate the required features. This model facilitated a systematic theoretical study of overlapping and early binding and suggested practical approaches for estimation of the impact of these mechanisms on application performance. In order to further facilitate the theoretical framework, this

chapter introduced new performance metrics, namely degree of persistence, degree of overlapping, segmentation efficiency, and degree of asynchrony. These metrics strive to provide a more accurate description and quantification of the complex interaction between the user application and the parallel system.

The results of the theoretical framework can be used by organizations that intend to improve the performance of their parallel codes through using overlapping and early binding. These organizations can estimate the performance benefit from the code optimizations and perform a feasibility performance-cost analysis, without actually performing these code optimizations. This can have a significant practical impact on the economics of these organizations.

The theoretical framework was applied to selected parallel algorithms. The performance of these algorithms was modeled with, and without, the studied performance enhancement mechanism. Then, the modeled performance of the two versions was compared. The performance gains from early binding and overlapping were expressed through the parameters of BOUM and the new metrics. The performance estimations of the studied algorithms are subjected to verification in Chapter V. The algorithms presented in this Chapter III are implemented in both versions and the actual performance of these algorithms is measured. The results from the measurements are used for proving the hypothesis of this study.

CHAPTER IV

EFFICIENT MPI IMPLEMENTAION FOR CLUSTERS OF WORKSTATIONS

This chapter presents a new MPI implementation that specifically targets clusters of workstations interconnected with high-speed networks. This presentation focuses on the part of the implementation that provides MPI communication services over Virtual Interface Architecture networks. The two physical networks used for development and validation of the implementation are Giganet cLAN (Giganet 1999) and ServerNet II (Compaq 2001). The MPI implementation described here has served as the foundation for the current generation of MPI/Pro products offered by MPI Software Technology, Inc. This implementation is referred to as MPI/Pro throughout this document. The author of this work has implemented the main MPI library functionality of MPI/Pro and co-developed the startup and build utilities for the MPI software development kit. This chapter first states the requirements and objectives of the new MPI implementation. Then, it reviews design considerations, performance trade-offs, and important architectural solutions. Finally, this chapter presents experimental results from point-to-point performance tests and from the NAS Parallel Benchmarks (Bailey et al. 1991).

4.1 Requirements and Objectives

The major objective of MPI/Pro is to provide an efficient, high-performance, scalable MPI implementation for clusters of workstations interconnected with high-speed networks. The following important requirements were considered in the software requirements specification phase of MPI/Pro:

- low CPU overhead,
- effective overlapping of communication and computation,
- asynchronous processing,
- independent message progress,
- optimized persistent mode of operation for enabling early binding,
- thread safety for enabling hybrid parallel models, and
- efficient multi-device architecture for supporting clusters of SMP nodes.

4.2 Design Considerations

The design of MPI/Pro has benefited from the experience and lessons learned from previous MPI efforts, especially MPICH (Gropp et al. 1996). The architectural solutions of MPICH were carefully studied in order to determine whether they could support the objectives of this new MPI implementation, and if so, how. It was assessed that this support would be insufficient in a number of important areas, and that adopting a design similar to MPICH would impede the achievement of the initial goals. This assessment led to the creation of a completely new design. This section provides a

discussion on important performance issues and reviews architectural features of MPI libraries that influenced the new design.

4.2.1 Completion Notification

Communication between processes in message-passing systems requires explicit participation of both the sender and the receiver. A successful message transfer requires a send transfer operation at the source node and a receive transfer operation at the destination node. The send operation is considered complete when the MPI library copies out the contents of the user source buffer to a system buffer or directly to the network. Similarly, the receive operation is complete when the library deposits the entire message into the user-specified receive buffer. The MPI standard (Message Passing Interface Forum 1994) specifies local and remote completion semantics of send operations and local completion semantics of receive operations. Local send completion indicates that the user process can safely reuse the send buffer. Local completion does not provide any guarantees about the status of the receiver process. In contrast, the remote send completion semantics guarantee that, when the send operation completes, the receiver will have begun the reception of the message. Local completion is used more often in MPI applications than is remote completion because it provides greater opportunities for performance optimization.

Message completion notification is a procedure for synchronization between the user processes and the MPI library. The MPI library uses completion notification to inform the user process about the finalization of the send or receive operations and the

availability of the participating buffers. When the user process requests a communication operation on a buffer, the “ownership” of this buffer is transferred to the MPI library until the moment when the operation completes. The completion status is propagated from the library to the user process through the completion-notification procedure. The period of time between the moment when the user request is submitted and the moment when the library signals completion is referred to as “completion synchronization.” Completion synchronization depends on the operating system, the network transport, and the MPI middleware architecture, but is independent of the actual message transmission time. Completion synchronization is one of the major factors for determining communication overhead.

There are two major forms of message completion notification: synchronous and asynchronous. Synchronous notification is usually implemented through polling on a synchronization object. This synchronization object can be a flag in memory that is signaled by the network controller through a system bus transaction or a kernel object whose status is checked by continuously calling a kernel routine. The type of synchronization object depends on the underlying communication layer, which can be kernel-based such as TCP/IP or user-level with operating system bypass such as VIA. Polling propagates the completion status of the requested operation to the user process with minimal delay, which results in reduced communication overhead. Low communication overhead is the major facilitator of low message-passing latency. On the other hand, polling causes the main CPU to operate in a busy-waiting mode during which

it cannot perform useful computation; hence, polling increases CPU overhead. As indicated earlier, high CPU overhead minimizes application's performance benefit from employing communication and computation overlapping.

MPICH is a typical representative of an MPI implementation that uses polling for message completion notification. Subsequently, most MPI implementations derived from MPICH bear the same architectural feature.

The asynchronous method for message completion is based on interrupts generated from the NIC and is implemented through kernel synchronization objects, such as semaphores, conditional variables, or events. The asynchronous method involves interrupt handling and introduces an extra context switch needed for signaling the synchronization objects. This context switch increases communication overhead. As a result, the asynchronous method for completion shows higher latency than the polling method. However, asynchronous completion reduces CPU overhead by releasing the CPU from immediately attending the completion procedure. The user thread that requests a communication operation is blocked on a kernel synchronization object and becomes ineligible for execution by the operating system until the blocking condition is met. The blocked thread does not use any system time. Meanwhile, the processor can be used to execute threads that perform useful computation. When the NIC completes the requested operation, it generates an interrupt to the CPU. Then, the CPU executes the interrupt handler. The interrupt handler is a module of the NIC driver, which in turn is a part of the

kernel. Finally, the operating system signals the synchronization object and releases the blocked user thread that becomes eligible for execution again.

MPI/Pro has a unique design that utilizes both methods of completion, and enables users to switch between the modes using a run-time flag. This option allows for a fair comparison between the two modes of completion notification and for studying their impact on latency, CPU overhead, overlapping, and overall application performance. This study shows that the increased of the asynchronous mode in respect to the polling mode has minimal or no impact on the performance of a large class of parallel applications and that the performance gain from overlapping outweighs this latency increase. This is among the major findings of this dissertation.

4.2.2 Message Progress

The MPI API provides a blocking and a non-blocking mode of communication (Message Passing Interface Forum 1994). The non-blocking API enables efficient asynchronous processing. This API consists of the following set of calls: the *MPI_Isend*, *MPI_Irecv*, the *MPI_Wait*, and *MPI_Test*. The *MPI_Isend* and *MPI_Irecv* group of functions are used for request submission, while *MPI_Wait* and *MPI_Test* are used for completion synchronization. Using the non-blocking API, the user process can submit a communication request to the MPI library and check for the completion of this request at a later moment. This allows the process to perform computation or another communication operation between the submission and the completion synchronization of the operation. In order to achieve the necessary semantics, the MPI standard requires that

the MPI library guarantee progress of communication associated with asynchronous requests. This requirement is often referred to as the "MPI Progress Rule" (Message Passing Interface 1994). The progress requirement ensures that once a message request is submitted, the communication this request specifies will be completed, regardless of user process' behavior. According to the Progress Rule implementation, there are two types of MPI library architectures, those with independent progress and those with polling progress.

MPI libraries with independent progress use an independently schedulable by the operating system progress agent. This agent can be implemented through asynchronous callback handlers or specially designated progress threads. In certain cases, the agent can be implemented through a combination of the NIC firmware (hardware thread) and the low-level messaging layer. Sandia Portals for Myrinet (Brightwell and Shuler 1996) is an instance of a system with a hardware thread used for message progress. The progress agent is executed independently of the call sequence of user processes. This guarantees that once a request is submitted, the communication associated with this request will be completed even when the user process does not make a subsequent call to the MPI library. MPI implementations that use independent progress comply with the so-called "strict interpretation" of the Progress Rule (Hebert et al. 1998). The independent progress engine usually relies on asynchronous completion notification as described in the previous section. MPI/Pro is an implementation that follows the strict interpretation. This is achieved through the use of a specifically designated progress thread that executes a

continuous message-processing loop. This loop makes a blocking call for checking the status of a global asynchronous event associated with incoming messages. The progress thread "sleeps" on the global event during periods when there are no incoming messages. While sleeping, the progress thread does not consume any CPU cycles.

Polling progress, on the other hand, requires that user processes make frequent calls to the MPI library in order to ensure timely progress of asynchronous requests. Progress is made only when the MPI library is called. Typically, MPI libraries with polling progress have a progress engine that is called within the majority of the MPI calls. Often, the progress engine is called even within MPI functions that do not require any communication. This technique is used to increase the frequency of calls to the progress engine. As a result, the execution time for these functions will vary widely, which will reduce the overall predictability of the MPI library.

Polling progress does not comply with the "strict interpretation" of the Progress Rule. Some MPI implementations with polling progress rely on coarse-grained, time-based interrupts to ensure progress of asynchronous transfers if the user process does not call the MPI library regularly. MPI libraries with polling progress often utilize polling completion notification for achieving low latency. However, similarly to polling notification, polling progress increases CPU overhead and lowers the degree of overlapping. The impact of the message progress on effective bandwidth is discussed in a subsequent section of this chapter.

4.2.3 Multiple Communication Devices

Clusters of multiprocessor workstations or servers offer two or more fabrics for communication between processors. Often, it is beneficial to use specifically designed operating system mechanisms for interprocess communication between processes on the same node. In order to utilize these mechanisms, the MPI library provides two "devices" (Gropp et al. 1996): one for intra-node communication and one for inter-node communication. These devices are often called SMP device and network device, respectively. The multi-device MPI architecture allows one process to communicate over all devices simultaneously. Multi-device MPI libraries enable the so-called "MPI everywhere" programming model.

In multi-device mode, MPI libraries with polling progress poll each device for communication events in a loop according to some policy (*e.g.*, round robin). Each device provides a different mechanism for propagating completion notification to the MPI library. Some devices require system calls; others require operating system bypass library calls. Slow devices may require relatively long times to check for completed events even when the event queues are empty. Since the completion check of a slow device is in a loop with all other devices, faster devices will experience increased overhead because of the slow device. Protopopov and Skjellum (2001) provide a study on MPICH's multi-device architecture and describe the negative interdependency between devices of different speeds when polling progress is used. Their specific attention is on the negative correlation between a slow TCP device and a fast SMP device.

MPI libraries with polling progress are often evaluated through ping-pong latency tests between two processes. However, this experimental setup uses only one device at a time, either SMP or network, and does not reveal the negative interdependency of slower devices on the overhead of faster ones. Tests that involve at least two devices operating simultaneously must be executed in order to reveal this interdependency.

As opposed to polling progress, independent progress does not introduce such interdependency between MPI devices; hence, faster devices are effectively "isolated" from slower devices. Since each device has an asynchronous progress agent that is independently schedulable, the MPI library progress engine does not need to check the devices for completed events; thus, unnecessary processing associated with passive devices is avoided. Furthermore, if there are enough computing and communication resources, the communication requests on different devices can be executed concurrently, which can allow for overlapping of two communication activities. It can be concluded that independent message progress leads to a more efficient architecture of multi-device MPI libraries than does polling progress. Independent message progress is also beneficial with respect to other I/O activities (*e.g.*, parallel file I/O) that are expected to progress asynchronously to message passing and computation.

4.2.4 Low Processor Overhead

Preserving processor cycles for useful computation is an important characteristic of communication systems. Since the primary goal of parallel processing is achieving fast computation, communication is viewed as pure overhead (a cost with no benefit). Two

major approaches for reducing communication overhead can readily be identified. The first approach is to optimize the communication system such that it provides faster data transfers. The second approach is to reduce CPU involvement in communication activities and to also reduce the effective impact of communication on overall execution time. The first approach emphasizes performance parameters such as low latency and high bandwidth. The second approach emphasizes factors such as low CPU overhead, communication and computation overlap, and reduced impact of synchronization. Combinations of both approaches are also possible.

The factors that determine the level of processor overhead of a communication system can be divided into hardware and software factors. The most important hardware factor is related to the capability of the NIC to perform communication without involvement of the central processor. The NIC of modern high-speed networks, such as Giganet and Myrinet, possess such capabilities. In contrast, traditional NIC's require the active participation of the main CPU in all communication and synchronization activities. The aforementioned software factors primarily depend on protocol stack efficiency and on message-passing middleware architecture. Thinner protocol stacks with no intermediate data copies facilitate low processor overhead. Similarly, middleware with asynchronous completion notification and independent progress minimizes CPU participation in communication activities, which leaves more processor cycles for useful computation. In contrast, polling notification and polling progress actively involve the main CPU in activities unrelated to the main user computation.

4.2.5 Latency

Reducing latency is a high priority of any communication system. Latency is critical for of short-message transfers. According to BOUM, the transmission time for a message of size m can be modeled as $T_c(m) = o + bm$, where o is the communication overhead and bm is the component that depends on the network bandwidth (b is the inverse of the bandwidth). Short messages are messages whose overhead component dominates the overall transmission time, $o > bm$. Alternatively, for long messages, $bm > o$. Consequently, short messages are sensitive to latency, and long messages are sensitive to bandwidth. Fine-grain data-parallel algorithms and highly synchronous applications that perform frequent barriers benefit most from low latency. Also, distributed shared memory systems that update memory across the network and applications that utilize one-sided communication are sensitive to latency.

The factors that affect latency include the hardware network capabilities, the protocol stack overhead, the completion notification scheme, and the message passing middleware progress method. Thinner protocol stacks using network interfaces featuring operating system bypass provide lower overhead than traditional multi-layer stacks. Asynchronous completion notification, henceforth referred to as blocking notification, and independent progress typically lead to a latency increase for short messages. Brightwell et al. (1999) refer to the combination of operating system bypass and independent progress as “application bypass.”

The combination of polling notification and polling progress offers the lowest latency. Usually the difference of overhead between polling and blocking notification depends on the operating system context switch facilities. Since the blocking notification method uses interrupts for synchronization, there is an extra kernel context switch for notification propagation to user process. Optimizing the interrupt handling procedures and context switches in operating systems may reduce the cost of the overhead incurred by blocking notification. Faster processors also reduce the synchronization overhead related to the operating system. The impact of CPU speed on latency in systems using both polling and blocking notification can be observed in the experimental results presented later in this chapter.

Latency is not generally considered a realistic measure of performance for applications that can overlap communication and computation. In addition, a large number of data-parallel applications predominantly exchange long messages, which are not latency sensitive. This work demonstrates that for a large class of medium to coarse-grained parallel algorithms the performance gain of overlapping outweighs the impact of increased message-passing latency that results from the use of asynchronous completion notification, independent message progress, and mechanisms for low processor overhead.

4.2.6 Bandwidth

Achieving peak bandwidth, as provided by the network data-link layer, is primarily impacted by host peripheral bus throughput and communication software efficiency. Presently, high-speed networks offer multi-gigabit-per-second links between

nodes, but the peripheral bus (*e.g.*, PCI) often limits the effective bandwidth to a fraction of peak. This leads to underutilization of the advanced performance features offered by high-speed networks. Eliminating intermediate data copies is the most important communication software feature for improving effective bandwidth. The VIA RDMA facility is a representative of a mechanism that enables the message-passing middleware to utilize protocol architectures with zero intermediate data copies, while freeing the main processors on both sides from participating in communication activities (Compaq, Intel Corporation, and Microsoft Corporation 1997).

Bandwidth is not affected by the increased communication overhead caused by asynchronous notification and independent message progress. On the contrary, both of these techniques facilitate higher effective bandwidth. First, asynchronous notification minimizes memory bus contention that results from simultaneous memory accesses generated by the CPU and by the NIC DMA engine. The DMA engine accesses the main memory through PCI transactions for moving data between the user buffer and the NIC local memory. Polling synchronization requires processor involvement that might lead to memory accesses that could collide with the NIC DMA transactions, thus reducing the effective bandwidth.

Second, independent progress guarantees timely transmission of long messages, which helps achieve top sustained bandwidth. Often, MPI libraries implement long message transfers using a three-stage rendezvous protocol (Gropp et al. 1996; Dimitrov and Skjellum 1999). This protocol requires the sender to initiate a synchronization

procedure, performed in the first two stages of the protocol, prior to forwarding the actual message to the receiver in the third stage. If the send request is asynchronous, an MPI library with polling progress can return to the user process before capturing receiver's acknowledgement. Then, the actual data transfer must occur when the user process makes a subsequent call to the MPI library. This call may be significantly delayed depending on the application algorithm. For example, an application may make a call to *MPI_Isend* with a long message, execute a long computation, and only then make another call to the MPI library (possibly to check the status of the request with *MPI_Test*). Depending on the timeline of the sender and receiver processes, MPI libraries with polling progress will perform the actual data transfer only when the latter MPI call is made. This will negatively impact the effective bandwidth as seen by the user application.

In contrast, MPI libraries with independent progress will react immediately to the confirmation from the receiver and send the message as soon as possible. The comparison of the two types of progress engines and their respective impact on effective bandwidth is depicted in Figure 4.1. This figure shows the rendezvous protocol that is typically used for transfers of long messages. The first two stages of the protocol exchange control messages, specifically, request to send (RTS) and clear to send (CTS). The RTS message informs the receiver about the size of the user data that is to be sent. The receiver acknowledges availability of buffer space to accept the requested message with CTS. The actual data transfer is performed in the third stage. In MPI/Pro, the third stage is performed by an RDMA Write operation. This operation is transparent to the receiver.

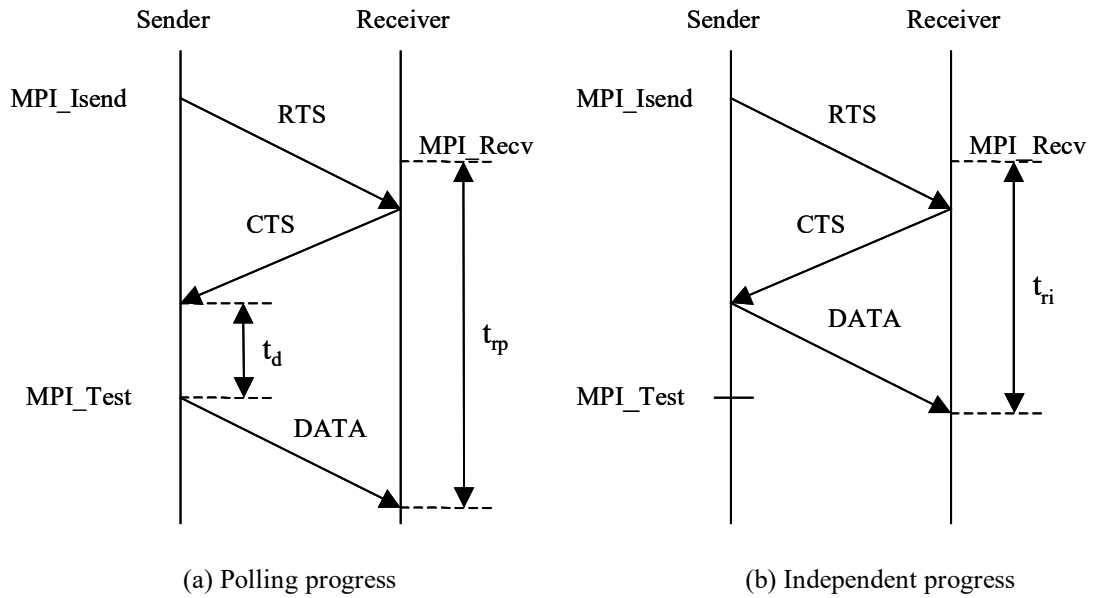


Figure 4.1 Impact of message progress method on effective bandwidth

If the size of the transferred message is m bytes, the effective bandwidth from the receiver's standpoint is computed as $BW = m/t_r$, where t_r is the period of time between the moment the receive request is posted (*MPI_Recv*) and its completion. For the MPI libraries with polling progress, this time is denoted with t_{rp}, whereas this time for the libraries with independent progress is denoted with t_{ri}. The time t_{ri} represents the sum of three components: the synchronization time between calling *MPI_Recv* and receiving the RTS message from the sender, the time for sending CTS, and the transfer time necessary for moving the m bytes of the message. Clearly, t_{rp} is longer than t_{ri} since $t_{rp} = t_{rp} + t_d$, where t_d is the time between the reception of the CTS message at the sender and the moment when the actual data transfer is initiated. The duration of t_d depends on the behavior of the user process. If the application calls *MPI_Wait* or *MPI_Test* in a tight

loop soon after *MPI_Isend* is issued, the time t_d could be negligible. In fact, this is what the typical ping-pong latency test does. However, if the user process performs long computation or some other communication or I/O, t_d could be significantly longer. Consequently, the effective bandwidth using polling progress, $BW_p = m/(t_{ri} + t_d)$, will be lower than the effective bandwidth using independent progress is used, $BW_i = m/t_{ri}$.

4.2.7 Persistent Mode of Communication

MPI provides an API for persistent mode of communication. This API consists of the following calls: *MPI_Send_init*, *MPI_Recv_init*, *MPI_Start*, *MPI_Wait*, *MPI_Test*, and *MPI_Request_free* (Message Passing Interface Forum 1994). The persistent API can be effectively used to take advantage of temporal locality in applications using early binding. Temporal locality is typically present in data-parallel and other regular parallel algorithms with iterative kernels.

As mentioned earlier in Chapter II, VIA mandates that all memory segments participating in data transfers must be registered. Memory registration is a high-overhead operation that requires time-consuming memory manipulations by the operating system. The use of persistent MPI operations enables reduction of the effective registration overhead by reusing registered segments for multiple communication transactions. Using the persistent MPI API is one of the main approaches for achieving effective early binding on clusters interconnected with VIA networks. Since MPI/Pro specifically targets VIA networks, the persistent API optimizations that minimize the impact of memory registration are considered to be a critical requirement.

4.2.8 Thread Safety

Support for multi-grain parallel processing through multithreading is one of the major design objectives of MPI/Pro. A number of widely used operating systems, such as Solaris, Linux, and Windows, offer efficient preemptive thread models for exploiting local parallelism and fine-grained concurrency. MPI/Pro targets all of these operating systems and also aims to provide mechanisms for efficient SMP processing. Thread safety is also a feature required by the independent message progress capability of MPI/Pro. Additionally, different categories of MPI users have emphasized the need for thread support in MPI for variety of purposes, among which are utilizing hybrid parallel models such as using MPI and OpenMP (Dagum and Menon 1998). An study of MPI in multithreaded environment is presented in Appendix B.

4.2.9 Efficient Use of VIA Features

MPI/Pro is designed to operate optimally on clusters of workstations interconnected with VIA networks. The key VIA features used in MPI/Pro are as follows:

- minimizing operating system involvement in critical communication operations,
- availability of hardware thread of control implemented by the VIA NIC,
- memory registration,
- remote DMA data transfers,
- large number of VI instances per NIC,
- scatter and gather modes for memory transfers,
- large MTU size (32 Kbytes),

- support of both synchronous and asynchronous modes of notification, and
- reliable, in-order packet delivery with no packet duplication.

4.3 Architecture

MPI/Pro incorporates several new architectural solutions that improve message-passing performance while also facilitating low processor overhead, higher degree of overlapping, asynchronous processing, and early binding. Below, some of the most important architectural solutions are identified.

4.3.1 Progress Thread

MPI/Pro uses a progress thread for implementing an independent, non-polling message progress. In most of the existing MPI implementations, progress of non-blocking or long messages is made only when user processes continuously call the MPI library. In contrast, MPI/Pro makes progress of all messages independently of the sequence of user process calls. Ultimately, MPI/Pro can complete a non-blocking send request even if the user never makes a subsequent call to MPI after the request is posted. The progress thread guarantees timely progress of asynchronous requests. Also, this thread is used to handle control traffic related to user-level flow control and the finalization protocol, which enables the library to gracefully handle unexpected terminations of MPI processes.

Using a library thread for message progress facilitates an asynchronous model for completion notification. The progress thread waits on a VI completion queue for

incoming communication events and does not consume any CPU cycles. When a packet arrives, it awakens the progress thread, which in turn processes this packet and takes actions corresponding to packet's content. Then, the progress thread goes to sleep again. A user thread may execute useful computation while the progress thread is blocked and awaits incoming messages.

4.3.2 Using RDMA for Long Transfers

MPI/Pro uses VIA RDMA operations for long data transfers. RDMA requires the active side to know the virtual address of the target buffer. MPI/Pro meets this requirement by the use of a rendezvous protocol for long messages. When a send request is posted, the sender forms an RTS control message specifying the size of the data to be sent. The receive side processes this message and replies with a CTS packet containing the address of the target receive buffer. Then, the sender initiates an RDMA transaction to transfer data. The last iteration of the protocol uses RDMA and is fully transparent to the target node. RDMA operations significantly reduce the CPU overhead. The CPU utilization of an MPI/Pro process transferring long-messages (64 kilobytes or more) is in the range of only 3-4%. Thus, MPI/Pro for VIA networks offers MPI applications the potential to really hide communication by overlapping it with computation.

4.3.3 Multiple Queues for Receive Requests

MPI/Pro uses multiple receive queues for posting and matching communication requests. In contrast, MPICH and most of its derivative implementations use only one

global pair of receive queues for posted and unexpected receive requests. In the single queue model, messages from all ranks that arrive before a matching receive request is posted are treated as unexpected requests and are queued on the *unexpected receive* queue. Similarly, receive requests from all ranks that are posted before a matching unexpected message has arrived are queued on the *posted receive* queue. Since the receive requests from all ranks are posted to the same queue, the matching procedure results in a linear search with asymptotic complexity $\Theta(NR)$, where R is the number of ranks (processes) participating in the MPI job, and N is the average number of outstanding requests on the receive queue per rank. As an illustration of the matching process and the asymptotic complexity of the single- and multiple-queue designs, the *MPI_Gather* collective operation is discussed. For simplicity, the case with $N = 1$ is reviewed. For the purposes of this example, a linear implementation of *MPI_Gather* is chosen. Generally, better asymptotic algorithms, such as binary or minimum spanning trees, are possible. It is assumed that the collective operation involves all R ranks.

In the chosen example, all $R - 1$ leaf processes send their messages to the root rank. If these messages arrive before the root rank submits the corresponding receive requests to the MPI library, the messages are queued to the *unexpected queue*. If the root process submits its requests before the arrival of the messages, the requests are queued to the *posted queue*. For simplicity, it is further assumed that the root node will be able to generate all receive requests and post them prior to the arrival of the first message from any of the $R - 1$ leaf ranks. Then, for MPI libraries with one receive queue, the length of

root rank's *posted queue* will be equal to $R - 1$. When the first message arrives (from any one of the $R - 1$ leaf ranks), the root rank performs a linear search in the *posted queue*. The search matches the source rank, communicator context ID, and user tag parameters of the incoming message to the requests in the queue. The asymptotic complexity of this search is $\Theta(R - 1)$ and the time for matching all incoming messages to the posted requests at the root rank will be $\Theta(R - 1) + \Theta(R - 2) + \ldots + \Theta(1)$ resulting in an overall asymptotic complexity of $\Theta(R^2)$.

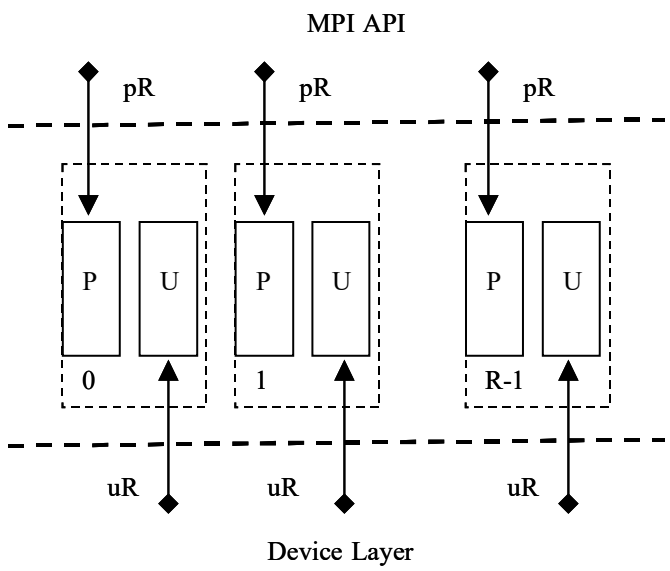


Figure 4.2 Schematic of multiple receive queues

MPI/Pro distributes the single pair of one posted and one unexpected queues to R such pairs, one per each rank (Figure 4.2). In the example described above, each queue will have a length of one request. Consecutively, the search for match will be $O(1)$. There will be R number of such searches and thus, the asymptotic complexity of the entire

collective MPI operation is $R\Theta(1) = \Theta(R)$. Hence, this new MPI/Pro queue architecture reduces the asymptotic complexity of the searches that match incoming and posted receive requests from $\Theta(R^2)$ to $\Theta(R)$. The multiple queue optimization of MPI/Pro affects all MPI communications, including point-to-point and control communications. Through this optimization, MPI/Pro achieves faster demultiplexing of the incoming messages. Therefore, it improves the overall performance of the implementation.

In Figure 4.2, P denotes posted request queues, U denotes unexpected request queues, pR denotes a posted receive request, uR denotes an unexpected receive request, and R denotes the number of ranks.

4.3.4 Synchronous and Asynchronous Completion Notification

MPI/Pro has the unique capability to offer users both synchronous and asynchronous methods for completion notification. Users select the desired method through a run-time switch. To the best of the author's knowledge, to date, MPI/Pro is the first and only MPI implementation that provides this capability. The advantages and disadvantages of the two methods for notification were independently reviewed earlier in this chapter. Using this feature, MPI/Pro offers users the flexibility to choose the optimal completion mode according to the aggregate requirements of the applications. Also, the dual-completion-mode capability is fundamental for understanding the complex software and hardware interactions that affect overlapping of communication and computation. The same application can be executed alternatively in both modes, which creates an experimental setting in which all elements are the same with the only exception being the

completion mode itself. This facilitates an objective study on the impact of completion notification on overhead, latency, and overlapping.

As described earlier, in asynchronous mode, the user thread is blocked on a kernel object for completion synchronization. This object is signaled by MPI/Pro's progress thread when the user request completes. In contrast, the synchronous mode of MPI/Pro eliminates the progress thread from the reception of short messages. Thus, if a process is expecting a message, it can poll directly for message arrival instead of blocking on the synchronization object. If the communicating processes are tightly synchronized, as in ping-pong latency tests, the message-passing latency can be reduced significantly at the expense of increased CPU overhead, as shown later in this chapter. In synchronous mode MPI/Pro, operates similarly to typical polling MPI libraries. However, as opposed to most polling-only implementations, MPI/Pro continues to use the progress thread even in the synchronous mode of completion. This thread is used for progress of long messages and also for handling control traffic. Thus, MPI/Pro eliminates the deficiency of other polling MPI implementations to require frequent calls to the library's progress engine for long messages. As was indicated earlier, independent progress facilitates higher effective bandwidth than does polling progress. This effect of message progress on effective bandwidth is often omitted in the performance analysis of message-passing libraries.

Figure 4.3 provides a classification of MPI implementations according to two dominant performance-defining factors: completion notification method and message progress scheme. According to completion notification, MPI libraries can be synchronous

(polling) or asynchronous (blocking). According to message progress, the libraries can be implemented with either independent or polling progress. Most commonly, MPI libraries use the "all-polling" architecture. MPI/Pro with its dual-mode completion notification scheme and use of progress thread covers almost the entire spectrum of combinations (the shaded blocks in Figure 4.3), with the exception of the case of polling progress with blocking completion, which has no practical meaning.

Polling notification **Polling progress** MPICH, MPI/Pro short protocol w/ polling mode	**Polling notification** **Independent progress**
Blocking notification **Polling progress** MPI/Pro long protocol w/ polling mode	**Blocking notification** **Independent progress** MPI/Pro blocking mode

Figure 4.3 Classification of MPI implementations

4.4 Summary of Features

This section summarizes some of the most important performance-oriented features of MPI/Pro, emphasizing its contributions to parallel processing on clusters and distinguishing it from other MPI implementations.

- MPI/Pro supports both synchronous and asynchronous methods of completion notification. Asynchronous notification is used to provide low processor overhead and enable a high degree of overlapping. The synchronous method is used for delivering low latency at the cost of increased processor overhead.

- MPI/Pro uses independent message progress based on a progress thread. Once a user request is posted, this request will be completed regardless of the behavior of the user process. MPI/Pro does not require user processes to call the library frequently in order to guarantee timely completion of asynchronous requests. Independent progress leads to increased effective bandwidth and also meets the requirements of the strict interpretation of the MPI Progress Rule.
- MPI/Pro is thread safe and enables hybrid parallel models using message passing between cluster nodes and multithreading for intra-node concurrency. This allows for exploitation of multi-grained parallelism.
- MPI/Pro optimizes the persistent mode of communication. VIA requires that all memory segments that participate in communications be pinned in physical memory. Memory pinning is a high-overhead kernel operation that causes context switches between the user process and the kernel. The negative effect of these context switches can be reduced if a memory segment is pinned once in physical memory and then reused multiple times. Thus, memory registration can be amortized over a large number of communications. MPI's buffer ownership semantics are compatible with these optimizations.
- MPI/Pro has a zero-copy protocol for data transfers of long messages through the VIA RDMA mechanism. After the initiation of the transfer, the DMA engine of the VIA NIC on the send node pulls data directly from the user buffer. Then, the NIC sends the message over the network to the receiver. At the receiver node, the

NIC drains the message from the network and, using its DMA engine, puts the data directly into the target user buffer. The entire procedure is transparent to the processors of both systems involved in the transfer. Zero-copy data transfers implemented with VIA RDMA achieve top sustainable bandwidth at minimum processor overhead.

- MPI/Pro uses multiple queues for posting receive requests. A pair of queues for posted and unexpected receive requests is associated with each MPI rank. This feature reduces the time for matching receive requests with incoming messages and optimizes message demultiplexing.
- MPI/Pro implements an optimized derived data type engine that provides efficient transfers of non-contiguous buffers.

4.5 Experimental Results

This section presents experimental results obtained using MPI/Pro on a variety of test cluster configurations. First, the test configurations and the notation used for referring to these configurations are introduced. The experimental results are presented in two groups: point-to-point performance and NAS Parallel Benchmarks. The results demonstrate the versatile features and performance capabilities of MPI/Pro, as well as an opportunity to study the impact of various cluster components on parallel performance.

4.5.1 Test Configurations and Experimental Methodology

The target environment of an MPI library is generally specified by three attributes: hardware platform, operating system, and network interconnect. These three attributes form a 3-D configuration space, which contains a large number of possible configuration combinations. Each configuration is a discrete point in this 3-D space. With the introduction of the MPI-2 Parallel I/O extensions, the target configuration space of MPI increases by one more dimension, specifically, the type of file system installed on the parallel platform. The MPI implementation presented in this work complies with the MPI-1.2 standard and does not offer MPI-2 extensions; therefore, the file system configuration attribute is not discussed.

The hardware platform attribute of a test configuration is defined by its processor architecture, processor clock rate, number of processors per machine, memory capacity, processor cache volume, memory system bus speed, and peripheral bus clock rate and width. A change in any of the specified platform components creates a new point along the platform axis of the configuration space. For example, a cluster with Intel architecture processors and a clock rate of 500 MHz is a different configuration from a cluster that has the same processor architecture but a clock rate of 800 MHz. The capacity of secondary storage is not considered to be an important performance-influencing component of the hardware platform for the purposes of this work.

The operating system attribute of the configuration space has less variability than does the hardware platform attribute. Presently, the operating systems commonly used for

building clusters are Linux, Windows, and Solaris. Only the operating system kernel version is of significant importance for Linux-based clusters. More recent kernels provide improvements in process context switching, thread support, and process scheduling.

The networking attribute of the configuration space has two aspects: physical network fabric and communication protocol. For this study, a change in any of these two aspects is viewed as a new point along the networking axis. For example, using TCP transport over Giganet and using the native VIA interface of the same physical network are considered different points in the network dimension. This separation in physical fabric and software protocol is justified by the substantial difference in communication performance, which is supported by the results presented below.

The goal of this section is to present experimental results that demonstrate the behavior of both MPI/Pro and the test cluster configurations by providing data that can be used for comparative performance analysis. For this purpose, at least two points are identified in each dimension of the configuration space while holding the other two attributes constant. This experimental methodology allows for a fair and precise analysis of the impact on performance of the attribute that varies.

The test configuration notation is based on labels that specify a point in the dimension along each of the three main configuration space attributes. Each configuration is annotated with a character string containing three labels, separated with hyphens, in the form *ppp-ooo-nnn*. The first label describes the hardware platform, the second label indicates the operating system, and the third label describes the network. In certain

configurations, more labels are used. These labels have different meanings for each case. For example, they may complement the networking label to present the combination of protocol and physical fabric. In other instances, the additional labels distinguish the results from round-trip-time latency and one-way latency measurements. In yet another case, they describe differences between the measurements obtained with MPI/Pro in polling and in blocking mode on the same configuration.

The platform labels used in this presentation are derived from the name of the hardware vendor, the specific computer model, or the name of the cluster. Specifically, three labels are used: *sag*, *dim*, and *ac3*. All clusters are built with Intel architecture nodes. The *sag* and *dim* clusters are operated at the main office of MPI Software Technology, Inc. in Starkville, Mississippi. The primary difference between the nodes of these clusters is the processor version and its clock rate: Pentium II @ 350 MHz for *sag* and Pentium III at 733 MHz for *dim*. The *ac3* cluster is the 64-node, 256-processor AC3 Velocity cluster operated by Cornell University (Cornell Theory Center 2001).

Two operating system labels are used: *win* for Windows and *lin* for Linux. The *sag* cluster runs Windows NT 4.0, while the *dim* cluster runs Windows 2000. Windows 2000 is also run on the *ac3* cluster. The *sag* and *dim* clusters are also dual-booted with RedHat Linux 6.2, kernel version 2.2. The primary network label has two values: *tcp* and *via*. The *tcp* label is used for TCP/IP-based communication over both 100 Mbit/sec switched FastEthernet and over Giganet. The *via* label is used for communication directly using the VIA interface of Giganet. When a comparison between *tcp*-based

configurations using different fabrics is presented, the fourth label is used to annotate the different physical fabric. For instance, the configurations *sag-win-tcp-eth* and *sag-win-tcp-gig* specify the same platform, the same operating system, and the same network protocol (TCP), but different physical media. When the physical fabric is not specified in the fourth label, it is always assumed that configurations with a *tcp* label use Ethernet and configurations with a *via* label use Giganet with its VIA interface. The complete specifications of all configurations used in this section and also in Chapter V are presented in Appendix C.

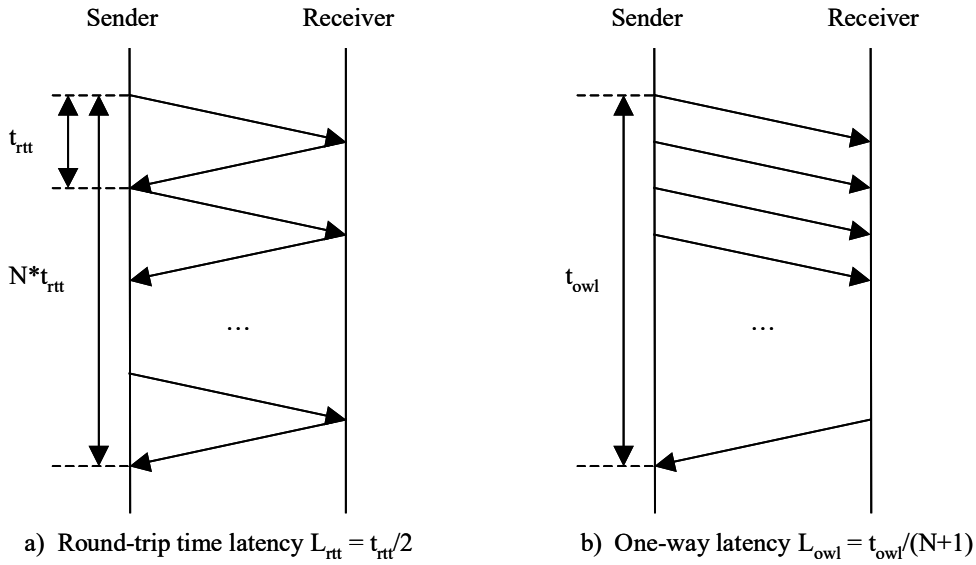


Figure 4.4 Methodology for measuring latency

4.5.2 Point-to-point Results

The point-to-point experiments are performed between two cluster nodes. The goal of these experiments is to measure the link performance of the communication

subsystem, as observed by the user processes. The results presented in this section are obtained from MPI test applications and show the combined performance attributes of the entire communication subsystem, including physical network, protocol stack, and MPI.

The experimental results are presented separately for latency and bandwidth. Latency is measured in microseconds and bandwidth is measured in megabytes per second. Two types of latency are defined: round-trip time latency and one-way latency. The round trip time latency is obtained by dividing the round-trip time of a message with a certain size by a factor of two. The round-trip time is measured by running a ping-pong test for each message size in a loop of N iterations and then dividing the entire communication time for the particular size by N. The one-way latency is obtained from a streaming test. One of the participating nodes sends a series of N messages of the same size to the receiver node. After the receiver node receives all N messages, it sends one message of the same size back to the sender. The one-way latency is obtained as the total time for the described procedure divided by $N + 1$. The schematic of the transfers and the time measurements for the round-trip time and one-way latencies are shown in Figure 4.4. In the latency graphs presented in this section, it is assumed that the curve represents round-trip time latency if the type of latency is not explicitly specified in the legend. All bandwidth results presented here are based on the round-trip time latency. This is a more conservative approach than measuring bandwidth based on the one-way latency.

The graph in Figure 4.5 presents the round-trip time latency on the *sag-win* configuration using four different combinations of the network configuration attribute.

Two of the network combinations use TCP transport, and the other two combinations use the VIA interface of Giganet. The TCP combinations are run with Ethernet (*tcp-eth*) and Giganet (*tcp-gig*). The VIA runs are executed with MPI/Pro operating in blocking mode (*via-blk*) and polling mode (*via-poll*).

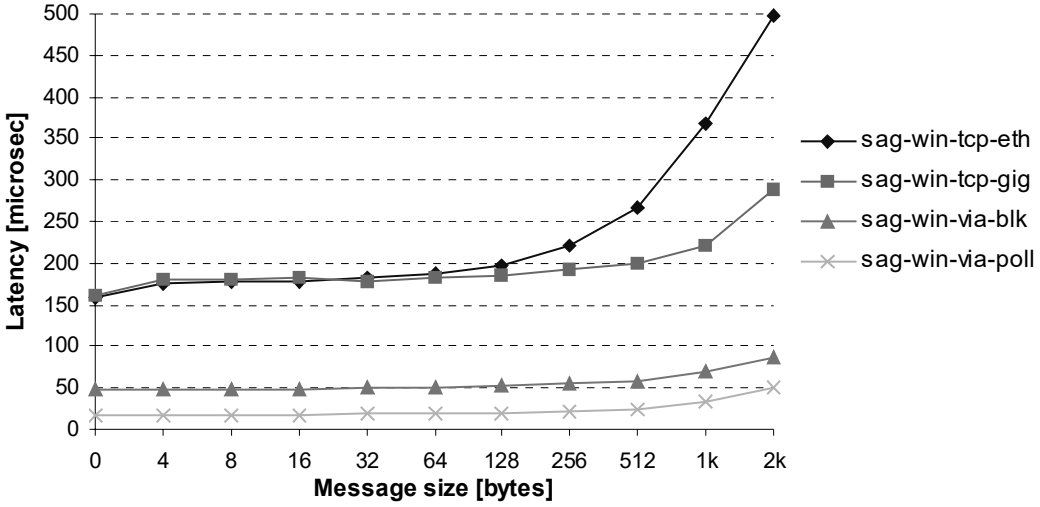


Figure 4.5 Round-trip time latency

Figure 4.5 provides several interesting observations. First, there is a significant difference in latency between the TCP transport and the transport layered based on VIA interface, even in the case when the TCP transport uses Giganet physical fabric (configuration *sag-win-tcp-gig*). TCP latency is almost an order of magnitude higher than the latency of VIA in polling mode. Since all other components of the parallel system are equivalent, this difference in latency performance can only be attributed to the software protocol stack. Clearly, protocols with a reduced number of abstraction layers and

featuring operating system bypass provide significant advantages in communication overhead compared to traditional protocols, such as TCP/IP.

The other observation made from Figure 4.5 is related to the impact of the network fabric on the latency of short messages. For this purpose, the *sag-win-tcp-eth* and *sag-win-tcp-gig* configurations are compared. For message sizes in the range [0, 256] bytes, the latency curves of the two configurations track each other with minimal difference. This demonstrates that the underlying network infrastructure, including the physical link layer, have minimal impact on latency. Latency of short messages primarily depends on the software overhead associated with message setup, transfer initiation, and completion notification. The importance of the completion notification method is demonstrated by comparing the two VIA configurations, namely *sag-win-via-blk* and *sag-win-via-poll*. The only difference between these two configurations is MPI's mode of message completion notification: blocking versus polling. The higher latency of blocking mode is explained by the higher communication overhead resulting from four activities that are not present in polling mode. These activities are as follows:

- executing the interrupt service routine of the VI Kernel Agent,
- signaling the synchronization kernel object,
- performing a process context switch between the kernel and the MPI process, and
- performing a context switch between the MPI progress thread and the user thread.

These additional activities account for the increase of latency from 21 to 48 microseconds for short messages. Dimitrov and Skjellum (1999) provide a detailed

breakdown of the time spent on each of these activities. The ping-pong test used for obtaining the round-trip time latency subjects the communication system to a type of traffic, which is not necessarily representative for the traffic patterns of the majority of data parallel algorithms (Dimitrov and Skjellum 2000). The ping-pong test traffic pattern can be viewed as an extreme point in a space of traffic patterns. The author of this work has proposed a streaming test for measuring one-way latency in order to identify the other extreme point in this space. Unlike the ping-pong test, the streaming test offers opportunities for hiding overhead through pipelining. Both of these extreme traffic patterns are rarely seen in real applications, but they can be used to obtain interesting insights about the behavior on the communication system under varying traffic conditions. Figure 4.6 provides a comparison between the round-trip time (*rt*) and one-way (*ow*) latencies of the *sag-win-via* configurations in blocking (*blk*) and polling (*poll*) modes.

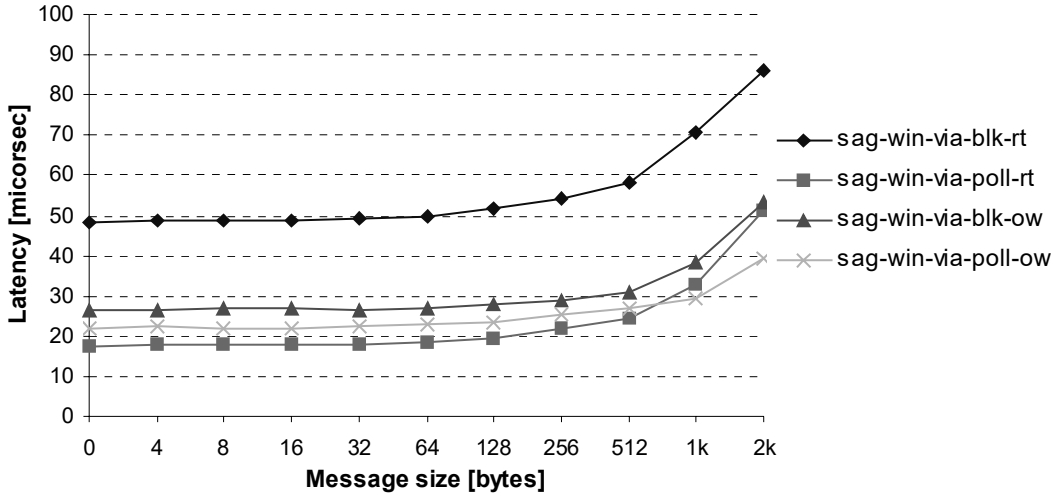


Figure 4.6 Comparison between round-trip time and one-way latency

First, the behavior of the blocking configurations *sag-win-via-blk-rt* and *sag-win-via-blk-ow* is analyzed. The curves representing the latencies of these two configurations demonstrate that in blocking mode one-way latency is almost two times lower than the round-trip time latency. This shows that careful scheduling of the traffic between two nodes can achieve a pipelining effect, which can substantially reduce the overhead associated with the blocking architecture of MPI/Pro. In fact, efficient utilization of pipelining, as in the streaming test (*sag-win-via-blk-ow*), can effectively reduce the latency of the blocking mode to approximately the same levels as the polling mode (*sag-win-via-poll-rt*). It is interesting to note that the polling mode of MPI/Pro does not show any latency improvement of the streaming test over the ping-pong test. Obviously, the polling mode of completion notification does not facilitate hiding overhead through the use of pipelining.

The graphs in figures 4.7, 4.8, and 4.9 demonstrate the impact of the operating system and the CPU speed on the latency of short messages. Figure 4.7 compares the *sag-win-via* polling and blocking configurations with the *sag-lin-via* polling and blocking configurations. The only difference between these two pairs of configurations is the operating system – Windows vs. Linux. As can be seen from the graph, the message-passing latency on Linux is higher than it is on Windows, especially in blocking mode. This difference can be explained by the higher cost of the process and thread context switches of Linux, as well as the slower kernel synchronization and mutual exclusion objects of this operating system. In fact, the POSIX threads *pthreads* package on Linux

(kernel 2.2) is implemented through the use of a full-blown process for each thread that the user creates. The only difference between "real" processes and processes that represent threads is that the latter share the same virtual address space, which allows them to access shared data structures. Linux does not have support for multi-threading in its kernel. In contrast, Windows threads are implemented in the kernel of the operating system. This enables Windows to provide more efficient thread context switching, as well as inter-thread synchronization and mutual exclusion. These are the main reasons for the lower overhead observed on Windows.

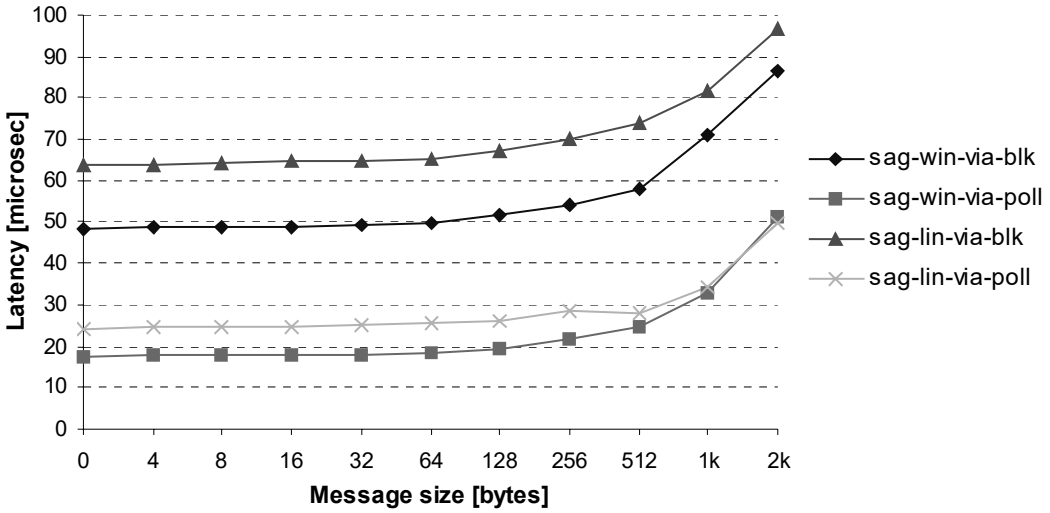


Figure 4.7 Comparison of latency on Windows and Linux

The impact of CPU speed on latency is depicted in figures 4.8 and 4.9. The *sag* cluster is built with single-processor workstations with Intel Pentium II processors operating at 350 MHz. The *dim* cluster is built with single-processor workstations equipped with Intel Pentium III processors operating at 733 MHz. The difference in CPU

clock rate is more than two times higher for the *dim* cluster. Also, the memory system bus clock rate of the *dim* cluster is increased from 100 MHz to 133 MHz.

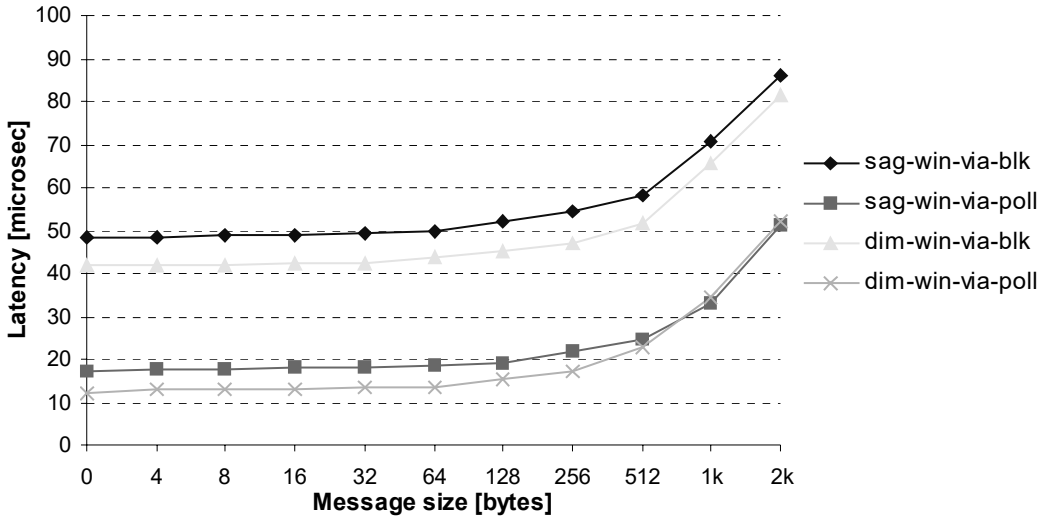


Figure 4.8 Impact of processor speed on VIA latency

Figure 4.8 demonstrates the latencies of MPI/Pro using the VIA interface in both polling and blocking modes across the *sag* and *dim* clusters. The faster CPU of the *dim* cluster reduces the short-message latencies of the blocking and polling modes by approximately seven and five microseconds, respectively. The absolute improvement for blocking mode is larger than the improvement for polling mode because the relative weight of the software overhead in the total communication time in blocking mode is higher than in polling mode and the increase of CPU speed affects most this overhead.

This conclusion is further supported by Figure 4.9, which depicts the impact of CPU speed on the latency of MPI/Pro using TCP transport. As shown earlier, the TCP transport latencies are much higher than the VIA latencies because of the heavier TCP/IP

protocol stack and the operating system involvement in communication. The use of the faster CPU leads to a latency reduction of more than 40% – from approximately 175 microseconds to approximately 105 microseconds in the message size range [0, 128] bytes. Figures 4.8 and 4.9 clearly show that the increase of CPU speed reduces the software overheads, which leads to a reduction of message-passing latencies, especially on configurations that operate with higher relative software overhead.

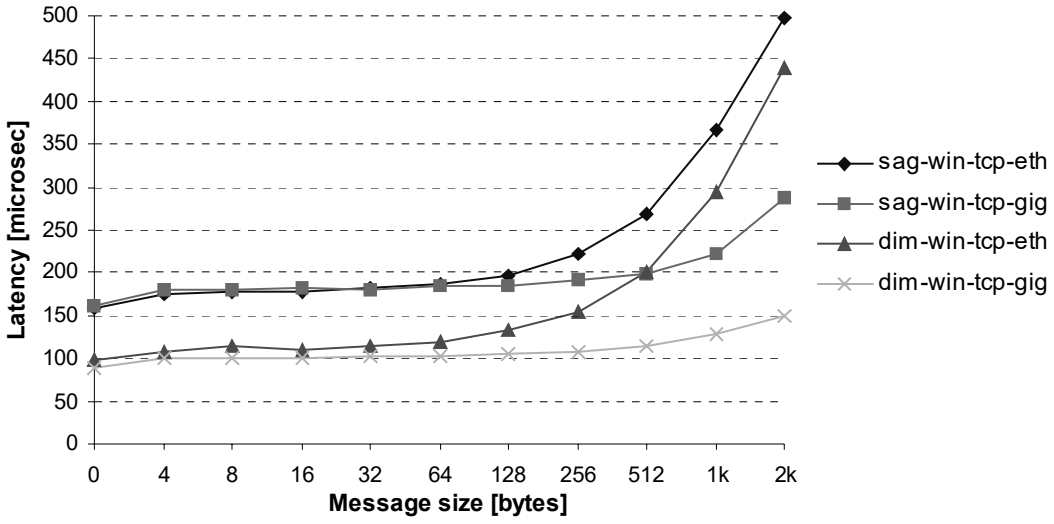


Figure 4.9 Impact of processor speed on TCP latency

The following series of graphs provide bandwidth experimental results. The first bandwidth graph is shown in Figure 4.10. This graph compares the bandwidth of MPI/Pro over the same transport in blocking and polling modes and over different transports in the same completion mode. Figure 4.10 shows that VIA transport (*sag-win-via-blk*) provides a factor of eight improvement in maximum sustainable bandwidth

compared to the TCP transport with Ethernet physical fabric (*sag-win-tcp-eth*). The maximum bandwidth of the two *via* configurations reaches 75 MB/sec and is only limited by the capabilities of Giganet system software and the peripheral bus throughput, rather than by the physical data rate of the network, which is 1.25 Gbit/sec. Using TCP communication over faster physical network (*sag-win-tcp-gig*) does not yield a significant improvement of peak bandwidth in respect to using TCP communication over slower network (*sag-win-tcp-eth*). This difference is only approximately a factor of 2.5 for the configurations in question, which is much smaller than the differences between the physical capabilities of Giganet and Ethernet. Evidently, the TCP/IP protocol stack does not exploit the network hardware resource efficiently. The main limiting factor that affects the bandwidth of the TCP transport is the intermediate data copies performed by the operating system when executing the software modules of the TCP/IP protocol.

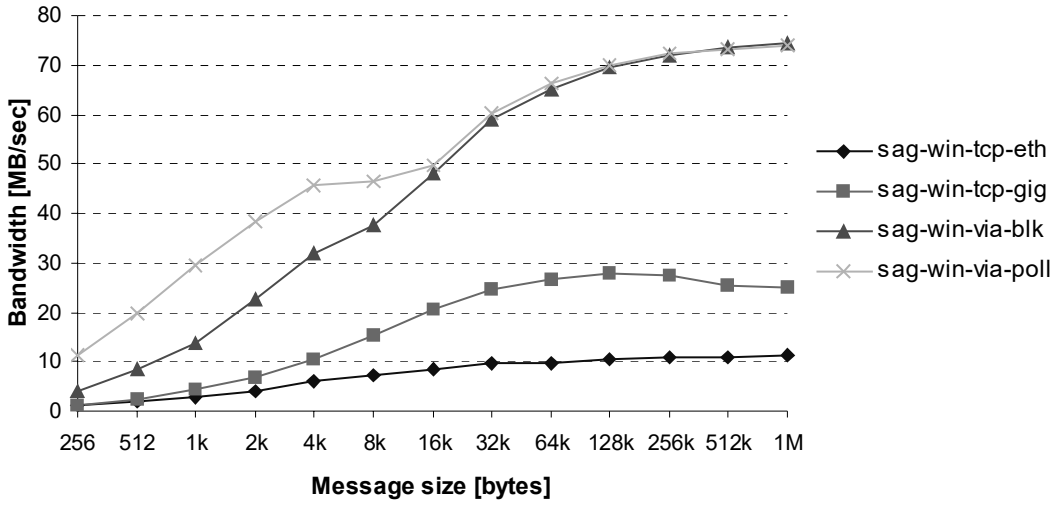


Figure 4.10 Impact of communication transport on bandwidth

In general, copies are the software factor that most significantly affects the effective bandwidth. The typical bandwidth-limiting hardware factors are physical network link rate, the network switching fabric structure, and the host peripheral bus throughput.

The comparison of the *sag-win-via-blk* and *sag-win-via-poll* configurations shows that the maximum bandwidth of the two is the same. Hence, the software overhead that causes substantial differences in short-message latencies has no impact on the bandwidth performance of mid-size and long messages. Using the representation of the communication time as defined by the BOUM model, the message transmission time for a message with size m is $t_m = o + bm$. Evidently, for long messages, the bandwidth factor bm is much larger than the overhead factor o and, consequently, $t_m \approx bm$. The difference in bandwidth for messages with sizes up to four kilobytes is attributed to the fact that the overhead factor o is dominant for these sizes and, consequently, influences the total message transmission time more than the bandwidth factor bm does. Therefore, a configuration with smaller overhead will result in a shorter transmission time, which naturally leads to higher effective bandwidth. This finding is similar to the well-known $n_{1/2}$ metric, where $n_{1/2} = overhead \times bandwidth = o/b$ defines the message size for which the overhead and bandwidth components of message transmission time are equal.

Figure 4.11 demonstrates the impact of the operating system on the maximum bandwidth. This figure compares the *sag-win-via-** configurations with the *sag-lin-via-** configurations, where the wildcard symbol "*" denotes either *blk* or *poll*. For messages of sizes up to two kilobytes, the Windows configurations show slightly higher bandwidth

than do the Linux configurations. This advantage is attributed to the lower latency of short messages on Windows, which was shown earlier in Figure 4.7. For the message sizes in question, the overhead still has a strong impact on the effective bandwidth. However, for message sizes beyond four kilobytes, the Linux configurations clearly outperform the Windows ones, reaching a peak at 95 MB/sec, a bandwidth increase of approximately 20 MB/sec. Since the cluster hardware, the physical network, the MPI library, and the test application are the same for all experiments presented in Figure 4.11, the significant difference in bandwidth can be attributed only to the operating system and the Giganet system software, which obviously interacts with the different operating systems in a different manner.

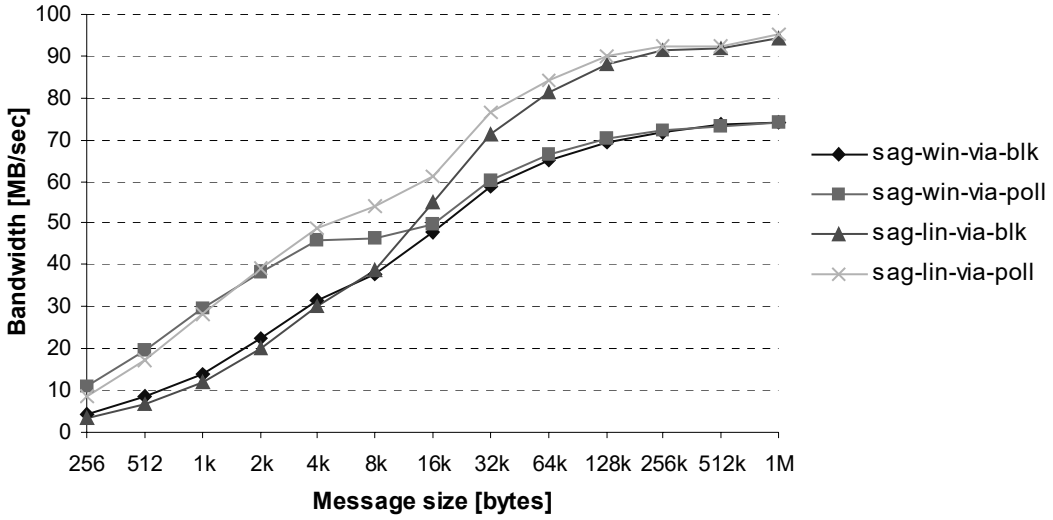


Figure 4.11 Impact of operating system on peak bandwidth

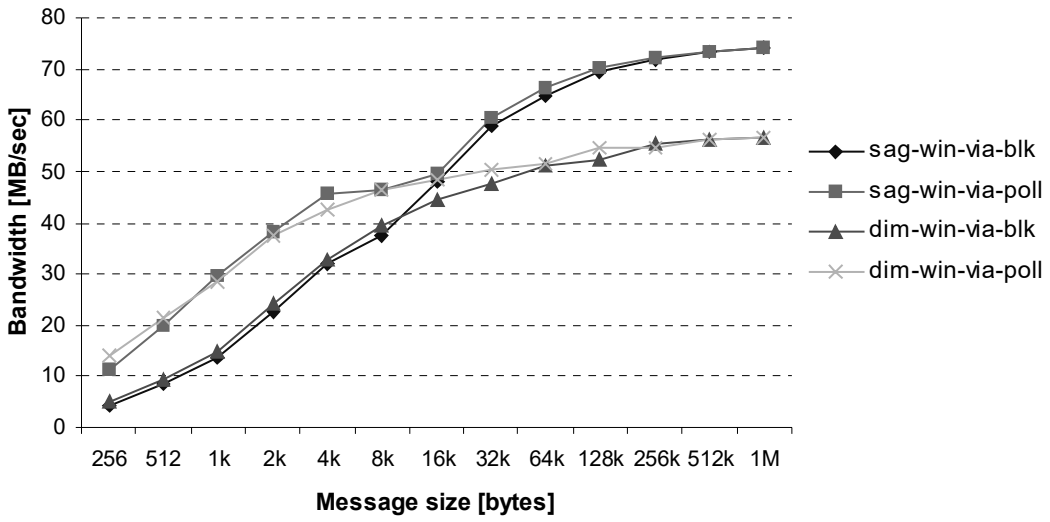


Figure 4.12 Impact of hardware platform on bandwidth

The last point-to-point performance graph, presented in Figure 4.12, shows the impact of the hardware platform on bandwidth. The important configuration variable in this figure is the platform label – *sag* and *dim*. As mentioned earlier, the *dim* cluster has a significantly faster CPU and faster memory system bus than does the *sag* cluster. Still, the bandwidth graph shows that the *sag* cluster achieves 20 MB/sec higher peak bandwidth. The bandwidth on the *dim* cluster reaches, at most, 57 MB/sec, which is only about half of the physical data rate of Giganet. Evidently, the limiting factor on this configuration is its PCI bus, which was confirmed with the vendor of this hardware. High-speed networks, such as Giganet and Myrinet, often surpass the PCI throughput capabilities of common computer configurations. Building a high-performance cluster with common-off-the-shelf components consequently requires careful evaluation of the solutions available on the market for such hidden performance bottlenecks.

4.5.3 NAS Parallel Benchmarks

In this subsection, selected NAS Parallel Benchmarks are used to evaluate the collective performance of the experimental configurations. Because both the *sag* and *dim* experimental clusters have eight single-processor nodes, the maximum number of processes in the collective experiments is eight. Performance results obtained on the *ac3* cluster with a larger number of processors are also presented. Although the total number of benchmarks in the NAS test suite is eight, experimental results from only three are presented here – CG, IS, and LU with classes A, A, and W respectively. The objective of this presentation is not to make an extensive study of the behavior of the NAS benchmarks on the experimental clusters. Rather, it is to reveal insights of how the architecture of the MPI implementation, specifically its message completion notification mechanisms, affects the collective performance of applications. The NAS Parallel Benchmarks suite contains application codes and kernels that are considered representative for the algorithms of a large class of numeric simulation applications.

An inspection of the source code of the NAS benchmarks shows that these codes use relatively straightforward approaches for implementing the communication sections of the algorithms and that minimal attention is given to performance enhancing techniques, such as overlapping of communication and computation and early binding. Therefore, the NAS benchmarks cannot exhibit any performance gains on systems that provide opportunities for such optimizations. Hence, the main goal of the NAS tests is to demonstrate the performance transparency of MPI/Pro operating in blocking mode. This

mode facilitates overlapping by providing asynchronous completion notification, independent message progress, and low CPU overhead. Proving this transparency will allow MPI/Pro to benefit codes that take advantage of overlapping and early binding while imposing no additional cost to codes that do not account for these optimizations. Figures 4.13, 4.14, and 4.15 present the results of the CG-A, IS-A, and LU-W tests on the *sag-lin*, *sag-win*, and *dim-win* configurations, respectively. In contrast to the point-to-point results, the NAS experiments do not strive to demonstrate the impact of configuration components on performance. As a result, study makes no comparisons between the *sag* and *dim* clusters or between Windows and Linux on the same benchmarks. Rather, each benchmark is executed on a different configuration and the impact of MPI/Pro's mode of completion on performance is investigated.

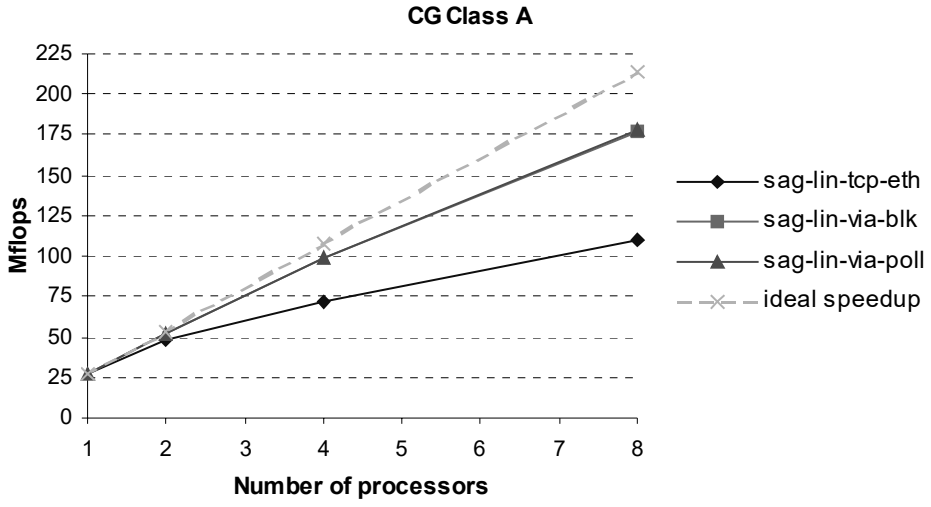


Figure 4.13 CG-A performance

The graph in Figure 4.13 presents the results from the CG-A experiments on the *sag-win* configuration with three types of communication stacks and hardware: *via-blk*, *via-poll* and *tcp-eth*. The results are reported in Mflops (millions of floating-point operations per second), which is the output of the NAS benchmarks. The CG benchmark is a non-stationary iterative solver that uses the Conjugate Gradient method. The Class A problem size solves a linear system with 14,000 equations. Single-precision floating-point coefficients are used; hence, here Mflops are single precision also.

From the graph in Figure 4.13, it is apparent that the overall performance is not affected by the message completion mode of MPI/Pro. Both the blocking and polling modes yield similar performance, which is 93% of the theoretical linear speedup for four processes and 84% of the linear speedup for eight processes. The performance of the tested configuration with TCP transport is significantly lower than with VIA – only 51% of the ideal speedup on eight processors. This shows that, for a larger number of processors, the scalability of the system that uses TCP communication will quickly degrade and the communication overhead will become the predominant factor in the overall execution time. This in turn will reduce the parallel efficiency of the system. In contrast, the configurations that use VIA communication demonstrate performance closer to the ideal speedup, which is a result of the increased communication capabilities.

It is interesting to note that although the performance of the VIA configurations is higher than the TCP configuration, the difference is not directly proportional to the difference found between the pure point-to-point link latency and bandwidth parameters

of TCP and VIA. As shown earlier, MPI/Pro using the VIA interface of Giganet yields approximately an order of magnitude higher bandwidth and an order of magnitude lower latency than does the TCP transport. However, the difference of the CG performance is less than a factor of two. This can be attributed to the fact that the CG benchmark is computationally bound and even significant improvement in absolute communication performance yields only moderate relative improvement in overall performance.

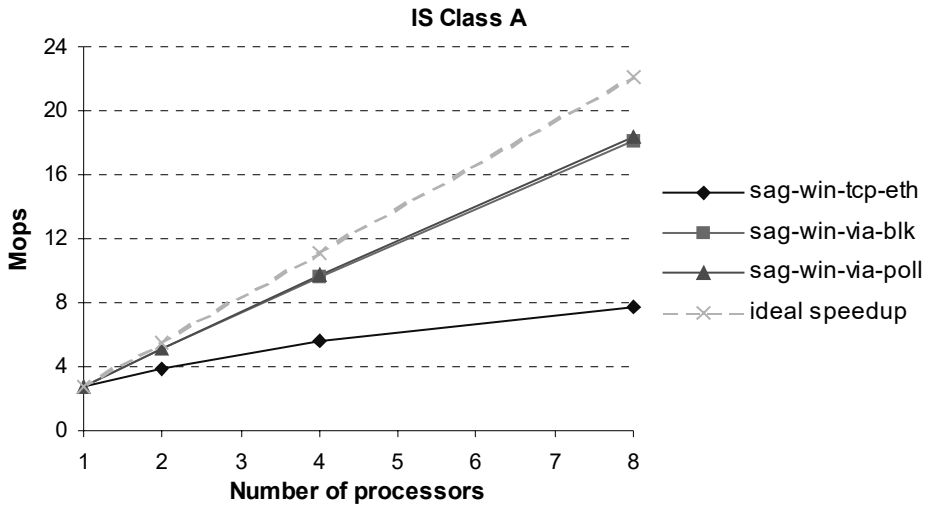


Figure 4.14 IS-A performance

Figure 4.14 demonstrates the results from the parallel integer sort (IS) benchmark with Class-A problem size (8,388,608 integers) on the *sag-win* cluster. Similarly to the CG-A experiment, the performance of the IS-A benchmark using the VIA configurations with MPI/Pro blocking and polling modes is the same. Both the blocking and polling configurations achieve about 84% of the ideal linear speedup for four and eight

processes. The IS-A performance of the TCP configuration is impacted more significantly by the lower bandwidth of TCP over Ethernet – it achieves only 35% of the ideal speedup. This is attributed to the higher communication intensity of the IS algorithm. This algorithm exchanges large messages and a significant portion of the entire execution time is spent on communication. Table 4.1 presents a breakdown of the traffic of IS with Class-A problem size, in bytes, for each process. The algorithm performs 11 iterations over an array of integers with a total size of W = 32 MB. The workload is distributed equally among all processors, *that is*, processor p receives a piece $W_p = W/P$, where P is the number of processors.

Table 4.1 Traffic pattern of IS-A benchmark

MPI function	**1**	**2**	**4**	**8**
MPI_Allreduce	4116	4116	4116	4116
MPI_Alltoallv	32M	16M	8M	4M
MPI_Alltoall	4	8	16	32

Three collective operations are performed in each iteration of IS: *MPI_Allreduce*, *MPI_Alltoallv*, and *MPI_Alltoall*. In each iteration, all processes distribute their local pieces to the other $P - 1$ processes and receive amounts also equal to W_p, totaling $2W_p$ bytes. This exchange is performed by the *MPI_Alltoallv* function. The total execution time T_p of the IS-A benchmark executed on the *sag-win-tcp-eth* configuration with four processes is 16.1 sec and the portion of this time spent on communication is T_{comm} = 10.4 sec. So, the ratio T_{comm}/T_p is equal to 0.65. Evidently, an improvement of the communication performance will have a sizeable impact on the total execution time.

Such improvement is demonstrated by the measurements obtained on the *sag-win-via-blk* configuration. The execution of the same IS-A benchmark on the *via* configuration resulted in a total execution time T_p = 8.7 sec and a communication time T_{comm} = 2.5 sec. The improvement of communication is more than a factor of four and has resulted in an overall reduction of the total execution time by 46% (from 16.1 sec to 8.7 sec). The communication improvement of the *sag-win-via-blk* configuration is primarily dependent on the superior bandwidth performance of the VIA/Giganet transport versus the TCP/Ethernet transport.

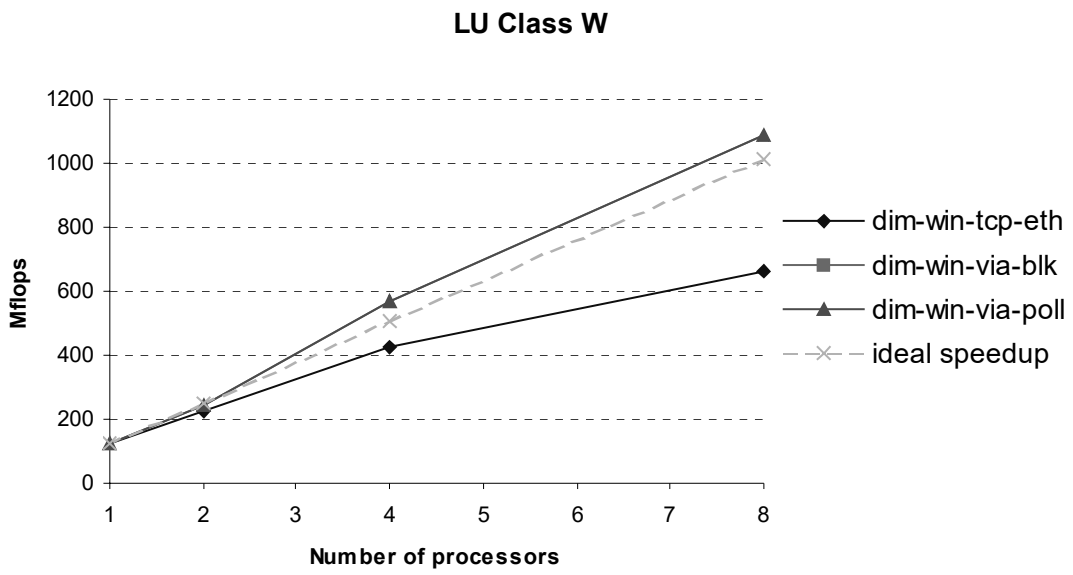


Figure 4.15 LU-W performance

The results from the LU-W benchmark obtained on the *dim-win* cluster are presented in Figure 4.15. The LU benchmark solves a system of linear equations with the

Navier-Stokes method based on Symmetric Successive Over-Relaxation (SSOR). In contrast to IS, the LU algorithm is not as communication intensive, which can be seen from by the TCP configuration performance. On eight processes, this configuration achieves 67% of the ideal speedup. This differs substantially from the performance of the TCP configurations obtained from CG-A and IS-A, which were 51% and 35%, respectively. The VIA configurations exhibit super-linear speedup as they show higher performance than the ideal linear speedup. This behavior is attributed to cache effects. The problem size processed by each processes is smaller for the eight-node run than is for the run on one or two processes. Smaller problem sizes evidently improve the cache behavior of the experimental cluster nodes. The benefits of the improved cache behavior outweigh the increased communication overhead when the number of processes grows. This ultimately leads to super-linear speedup.

The comparison of the blocking and polling VIA configurations in Figure 4.15 again shows that the performance of the two is the same. The three graphs that presented the performance of the CG-A, IS-A, and LU-W benchmarks demonstrated that regardless of the type of operating system and hardware platform, the collective performance of parallel algorithms that use MPI/Pro with the VIA interface of Giganet is not influenced by the notification mode for message completion. Although the blocking mode leads to an increase of short message latency, this increase does not translate into performance losses. From this, it can be concluded that the MPI/Pro optimizations that facilitate a high degree of communication and computation overlapping in blocking mode are

"transparent" to the performance of the NAS Parallel Benchmarks. Consequently, applications that do not utilize these optimizations (as the NAS benchmarks) will not exhibit performance degradations, while applications that exploit these performance-enhancing techniques will achieve performance gains. The actual performance benefits of the optimizations that facilitate overlapping and early binding are demonstrated in Chapter V.

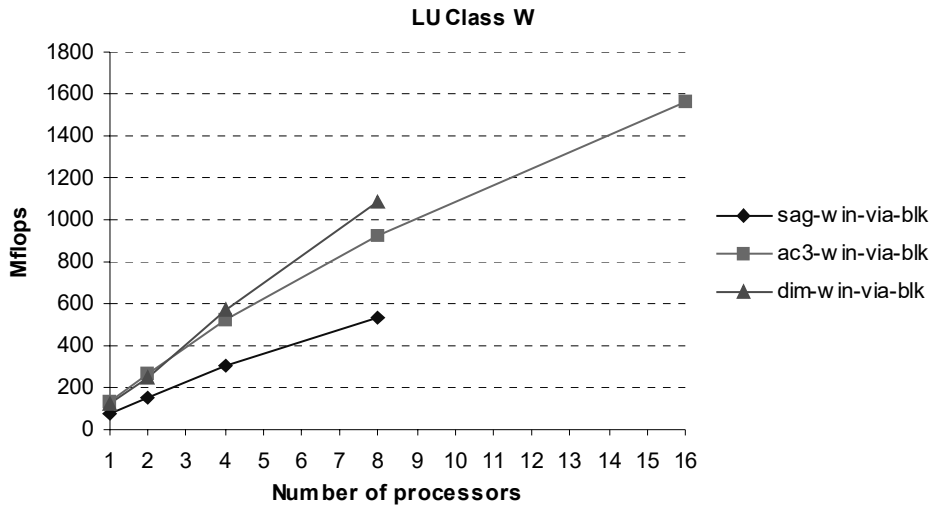


Figure 4.16 Impact of CPU speed on LU-W performance

Figure 4.16 provides a comparison of the absolute performance of the LU-W benchmark on the three experimental clusters running Windows and using the Giganet VIA interface. The tests were obtained with MPI/Pro operating in blocking mode. The test on the *ac3* cluster was executed on 16 processors. The graph shows that the absolute performance depends on the processor speed, which affects the slope, but not the shape,

of the curves. This indicates that scalability is primarily influenced by the communication capabilities of the parallel system.

4.5.4 Summary of Results

This section presented experimental results from a series of point-to-point and collective tests on a variety of cluster configurations using MPI/Pro. The emphasis of the point-to-point tests was on the impact of parallel systems' components on short-message latency and bandwidth. The experiments identified a number of factors that influence point-to-point performance. Among these factors are physical network medium, communication protocol stack, CPU speed, PCI bus throughput, operating system, message progress mechanism, and notification completion method of the message-passing middleware.

The collective performance results were obtained from selected NAS benchmarks. The focus of the collective experiments was on determining the impact of the MPI completion notification mode on the overall performance. It was demonstrated that although the blocking mode of notification causes higher short-message latency, its overall performance is the same as the polling mode. This is an important conclusion that supports the hypothesis that the message-passing middleware of a parallel system can provide optimizations that facilitate efficient overlapping of communication and computation at little or no cost to the applications that do not benefit from overlapping. The practical implication of this conclusion is that users can benefit from advanced message-passing middleware without losing performance of legacy applications.

4.6 Conclusions

This chapter presented MPI/Pro – a new MPI implementation developed by the author of this work. MPI/Pro is specifically designed to provide optimal performance on clusters of workstations interconnected with high-speed VIA networks such as Giganet and ServerNet. The design of MPI/Pro emphasizes low processor overhead, independent message progress, overlapping of communication and computation, early binding, asynchronous processing, internal concurrency, and multithreading. Also, MPI/Pro offers users the unique capability to tailor the desired mode of message completion notification to their specific needs. This capability was extensively used during the experimental phase of this work to demonstrate the impact of blocking and polling modes of completion on the point-to-point and application performance.

First, this chapter presented the requirements and design considerations used for the development of MPI/Pro. Then, important architectural solutions and key MPI/Pro features were summarized. Finally, the chapter presented experimental results from point-to-point and collective performance tests. The point-to-point tests provided insight to the numerous factors that determine the latency and bandwidth performance of a parallel system. The collective tests showed the impact of communication on overall performance and demonstrated that the blocking completion notification method of MPI/Pro does not negatively impact overall application performance. This hypothesis was supported by several experiments, which indicated that message-passing middleware facilities for efficient overlapping could be implemented in a performance transparent manner. Hence,

representative applications that do not take advantage of overlapping will not lose performance, while applications that employ overlapping can gain performance.

As a concluding remark of this chapter, MPI/Pro has been used on two large-scale clusters for reaching performance levels that qualify these clusters among the Top 500 supercomputers in the world. This ranking uses performance numbers obtained by the parallel LINPACK benchmark. The first cluster is the 256-processor AC3 Velocity Dell/Windows cluster at Cornell University, and the second cluster is a 256-processor SGI/Linux cluster operated by the NSF Engineering and Research Center at Mississippi State University.

CHAPTER V

VALIDATION OF HYPOTHESIS

This chapter presents experimental results and analyses that validate the hypothesis of this dissertation. These results were obtained from experimental executions of the algorithms implemented according to the description in Chapter III. This chapter first specifies the objectives of the experimentation and validation procedures, as well as the experimental methodology. The presentation of the experimental data is divided into two sections. The first section focuses on the results that demonstrate the impact of early binding on communication performance. The second section focuses on the results that show the impact of communication and computation overlapping on overall performance. Also, analyzed is the effect of the message-passing middleware architecture on the capability of a parallel system to support asynchronous processing and overlapping. Finally, a summary and interpretation of the experimental results are presented.

5.1 Objectives

The experimental process presented in this chapter has two main objectives. The first objective is to demonstrate, through empirical data, that early binding and overlapping of communication and computation are valuable sources of performance enhancement that can be successfully applied to a wide range of parallel

applications. Also, within the scope of this objective is to demonstrate the impact of the message-passing middleware architecture on the effectiveness of early binding and overlapping. The second objective is validation of the accuracy of BOUM and demonstrating the descriptive power of the newly introduced performance metrics. Validating the accuracy of BOUM in turn has two further aspects. The first aspect is accuracy of estimating the absolute performance of parallel algorithms on a specific platform. The second aspect is validating the accuracy of the model to estimate the performance gain of early binding and overlapping.

5.2 Experimental Methodology

This section describes the experimental methodology for obtaining and presenting the results that lead to validation of the hypothesis. This methodology is described separately for early binding and overlapping.

5.2.1 Early Binding

The methodology for early binding is based on the theoretical framework presented in Chapter III. This framework provides a formal definition of early binding as a software mechanism and introduces the degree-of-persistence metric for measuring the capacity of a parallel system to support effective early binding. The experimental methodology for early binding includes the following steps:

- selecting parallel algorithms with communication structure that suggests effective use of early binding,

- deriving expressions for execution time using BOUM parameters,
- applying early binding to the algorithms and expressing the expected performance gain through the use of BOUM parameters the degree-of-persistence metric,
- implementing the selected algorithms with MPI/Pro,
- measuring the values of the degree-of-persistence metric for the message sizes that are used in the algorithm,
- performing test runs without early binding on the target configurations,
- validating observed performance against the performance predicted by BOUM,
- performing tests with early binding on the same target configurations,
- observing the performance improvement (or degradation) when compared with runs without early binding, and
- comparing the actual performance gain against the prediction made with BOUM.

5.2.2 Overlapping of Communication and Computation

The experimentation and validation methodology for overlapping is similar to the methodology used for early binding. However, since the scope of the thesis with respect to overlapping is broader than its scope for early binding, the experimental procedure for overlapping also requires test executions on message-passing middleware with different architectures, specifically message completion notification and message progress. In addition, this work has thus far defined three metrics needed for accurate representation of the complex interactions between the software and hardware components of parallel systems and their overall effects on overlapping. Practical procedures for obtaining the

values of these metrics and measurements on the test cluster configurations are presented. The experimental methodology for overlapping includes the following steps:

- selecting an algorithm with communication and computation structure that suggests effective use of overlapping,
- deriving a theoretical expression for the performance of the algorithm with BOUM,
- applying overlapping optimizations to the selected algorithm and expressing its performance gain using BOUM and the metrics specified in Chapter III,
- measuring the values of the metrics on the test clusters,
- implementing the selected parallel algorithm with MPI/Pro,
- performing experiments with the non-optimized algorithm,
- validating the estimation of the execution time obtained with BOUM,
- performing experiments with the optimized algorithm using a variable number of overlapped segments (with asynchronous mode of completion notification and independent message progress),
- comparing the experimental results to the modeled estimations and validating the capability of the model to accurately describe the effects of overlapping,
- performing experiments with the optimized algorithm on the message-passing middleware using polling completion notification and polling progress, and
- validating the hypothesis that the MPI library architecture with blocking completion and asynchronous progress achieves higher performance gains

through overlapping than does the architecture with polling notification and polling progress.

5.3 Experimental Cluster Configurations

The experimental results presented in this chapter have been obtained from tests performed on the cluster configurations specified in Chapter IV and described in greater detail in Appendix C. Also, a special, label-based notation again has been used (introduced in Chapter IV) in order to specify the configuration characteristics of the test clusters, specifically: hardware platform, operating system, and network. In addition, this notation specifies the completion notification method used by the message-passing middleware – asynchronous (blocking) or synchronous (polling).

5.4 Obtaining BOUM Parameters

The parameters of BOUM were defined in Chapter III. These parameters are used in the expressions for the parallel execution times of the studied algorithms. The parameters of BOUM are as follows:

- t_c – time for basic unit computation,
- o – message-passing overhead, and
- b – inverse of the effective bandwidth.

For purposes of accuracy, this presentation elects to measure t_c for each algorithm individually, based on the sequential implementation of the particular algorithm. There are two major reasons for this choice. The first reason is that the time for a basic unit

computation is algorithm-specific. For instance, an algorithm for integer sorting performs comparison and assignment of integer values as its basic unit computation, while an algorithm for matrix-vector multiplication of double-precision elements performs multiplication and addition of double precision floating-point values. Evidently, the cost in time of the basic operation of the two algorithms may differ substantially, and therefore it must be modeled differently.

The second reason is related to the accuracy of the asymptotic-complexity analysis. This analysis reflects only the highest order of basic operations and often ignores constant multiplicative factors. For example, the asymptotic complexity of the parallel FFT is represented as $\Theta(n\log n)$, meaning that the highest order of basic computations is proportional to $n\log n$. However, the actual algorithm performs on average a total of five multiplications and additions for each element, so a more precise representation of the complexity of the FFT algorithm would reflect this fact by using an expression proportional to $5n\log n$. In order to avoid such specifics, the approach of this work is to accept that the basic unit computation is an aggregate "basic operation" that includes smaller units of work, such as additions and multiplications. Consequently, the approach for measuring t_c is as follows:

- Determine the asymptotic complexity of the sequential algorithm $\Theta(f(n))$.
- Measure the execution time of the sequential algorithm T_s.
- Determine t_c as $t_c = T_s/f(n)$.

This measurement approach reflects the specifics of the particular algorithm and the fact that the asymptotic complexity analysis ignores constant factors or factors with lower order than the highest order factor. This approach provides for an accurate measurement of the cost of the basic unit computation. Following this procedure, the values of t_c will be determined individually for each combination of algorithm and target configuration.

The measurement procedures of the communication-oriented BOUM parameters o and b are based on point-to-point ping-pong test, as commonly accepted in parallel processing and networking. Such experiments have already been presented in Chapter IV. The procedure is performed on each test configuration and is independent of the studied algorithm. Hence, o and b are viewed as configuration-specific parameters in contrast to t_c, which is specific to both the algorithm and the target configuration. An important element of the procedure for determining the communication parameters is the accurate measurement of the overhead. As mentioned in Chapter III, assuming that the overhead is constant for all sizes may be too optimistic. Therefore, the measurement procedure is based on the definition of setup/finalization overhead o_s and initiation/completion overhead o_i, as specified in Chapter III. The setup overhead o_s will be measured with the persistent MPI API. An MPI test program that creates and frees a communication request for all tested sizes is used for this purpose. The time for creation and releasing of the persistent request is recorded and then averaged over one send and one receive experiment. The pseudo code of the test program for measuring o_s is shown in Figure 5.1.

```
tb = time()
```

```
init_send(<args>, size = m, sreq)
init_recv(<args>, size = m, rreq)
free_request(sreq)
free_request(rreq)
te = time()
os = (te - tb)/2
```

Figure 5.1 Pseudo code for measuring setup overhead

The initiation overhead o_i is determined to be the duration between the moment when a persistent request for a zero-length message is started and the moment when the request is completed. The procedure for measuring o_i is presented by the pseudo code in Figure 5.2. The requests used for measuring o_i are also created with the persistent API. These requests correspond to messages in a ping-pong test. The time between the initiation of the first request (**sreq**) and the completion of the second request (**rreq**) at the process with rank zero is in effect the round-trip time (RTT), including synchronization and completion notification times. The initiation overhead is determined to be ½ of RTT.

The total overhead o for a given message of size m is found as the sum of the setup overhead o_s for this size and the zero-length request initiation overhead o_i: $o(m) = o_s(m) + o_i$. According to this procedure, the total message overhead (in its initiation overhead component) accounts for the time associated with passing control information in the message-passing middleware's system-specific header. Other parallel programming models that make use of overhead and throughput parameters often ignore this overhead.

The procedure for measuring the inverse of the bandwidth b is similar to the pseudo code from Figure 5.2, and is presented in Figure 5.3. After the initiation overhead

o_i is determined, the procedure in Figure 5.3 is executed for a given message size m for which the parameter b is to be determined. The time t_e - t_b is the round-trip time RTT(m) for this message. Then, the time t_{bm} that depends on the effective bandwidth is found as t_{bm} = RTT(m)/2 – o_i. Finally, b is found as $b = t_{bm}/m$.

```
if (rank == 0) peer = 1 else peer = 0
init_send(<args>, size = 0, dst = peer, sreq)
init_recv(<args>, size = 0, src = peer, rreq)
if(rank == 0)
tb = time()
      start(sreq)
      wait(sreq)
      start(rreq)
      wait(rreq)
te = time()
else if(rank == 1)
      start(rreq)
      wait(rreq)
      start(sreq)
      wait(sreq)
endif
free_request(sreq)
free_request(rreq)
oi = (te - tb)/2
```

Figure 5.2 Pseudo code for measuring initiation overhead

In summary, the procedure for measuring o and b for a given message size m is as follows:

- Perform the experiment for determining the setup overhead o_s for this message, as shown in Figure 5.1.
- Perform a ping-pong test for measuring RTT for a message with zero length and determine the initiation overhead o_i, as shown in Figure 5.2.
- Determine the overhead $o = o_s + o_i$.

- Determine the bandwidth time t_{bm} for this message and compute $b = t_{bm}/m$, as suggested by the pseudo code in Figure 5.3.

```
if (rank == 0) peer = 1 else peer = 0
init_send(<args>, size = m, dst = peer, sreq)
init_recv(<args>, size = m, src = peer, rreq)
if(rank == 0)
tb = time()
      start(sreq)
      wait(sreq)
      start(rreq)
      wait(rreq)
te = time()
else if(rank == 1)
      start(rreq)
      wait(rreq)
      start(sreq)
      wait(sreq)
endif
free_request(sreq)
free_request(rreq)
rtt = te - tb
tbm = rtt/2 - oi
b = tbm/m
```

Figure 5.3 Pseudo code for measuring BOUM bandwidth parameter

Below, the results for *o* and *b* on the selected experimental cluster configurations are presented. These results are listed in tables for message sizes in the range [0 – 1MB]. In these tables, the first column represents the message size in bytes. All times are expressed in microseconds for clarity of presentation. The bandwidth component *bm* of the message transmission time is shown next to the parameter *b*, in order to enable easier comparison with the overhead *o*. Each table represents the value of *o* and *b* for a particular test configuration with MPI/Pro operating in blocking and polling mode of completion notification.

Table 5.1 Measurements of BOUM parameters on *sag-win-via* configuration

m [byte]	blocking			polling		
	o [μsec]	b [μsec/byte]	bm [μsec]	o [μsec]	b [μsec/byte]	bm [μsec]
0	57.3	n/a	0	25.2	n/a	0.0
4	57.3	0.025	0.1	25.2	0.015	0.1
8	57.3	0.025	0.2	25.2	0.014	0.1
16	57.4	0.027	0.4	25.2	0.015	0.2
32	57.4	0.028	0.9	25.6	0.014	0.5
64	57.6	0.027	1.7	25.2	0.014	0.9
128	57.7	0.026	3.3	25.2	0.015	1.9
256	57.8	0.027	6.9	25.2	0.018	4.6
512	70.9	0.019	9.5	39.3	0.014	7.3
1024	74.4	0.024	24.1	39.0	0.017	16.9
2048	70.8	0.018	37.5	38.9	0.016	33.7
4096	71.8	0.019	76.7	39.7	0.018	73.2
8192	82.9	0.023	185.3	51.0	0.024	196.1
16384	85.4	0.017	272.0	53.3	0.019	303.9
32768	94.0	0.015	480.9	58.4	0.015	502.4
65536	103.5	0.014	922.9	70.0	0.014	920.1
131072	127.2	0.013	1754.3	95.8	0.013	1753.3
262144	183.3	0.013	3400.4	145.3	0.013	3445.0
524288	293.6	0.013	6774.2	260.7	0.013	6796.0
1048576	520.0	0.013	13424.8	489.1	0.013	13470.5

Tables 5.1, 5.2 and 5.3 demonstrate the dependencies of the overhead and bandwidth on the hardware platform, operating system, and the MPI completion-notification mode. These dependencies are similar to those discussed in Chapter IV. This discussion was specifically focused on the relationships between overhead and hardware platform, overhead and operating system, overhead and MPI completion notification mode, and bandwidth and hardware platform. Please refer to Chapter IV for more detail regarding these relationships.

Table 5.2 Measurements of BOUM parameters on *sag-lin-via* configuration

m [byte]	blocking			polling		
	o [μsec]	b [μsec/byte]	bm [μsec]	o [μsec]	b [μsec/byte]	bm [μsec]
0	75.9	n/a	0.0	31.3	n/a	0
4	75.9	0.006	0.0	31.4	0.014	0.1
8	75.9	0.007	0.1	31.3	0.014	0.1
16	75.9	0.007	0.1	31.5	0.014	0.2
32	75.9	0.006	0.9	31.3	0.015	0.5
64	75.9	0.006	0.4	31.3	0.014	0.9
128	75.9	0.007	0.9	31.4	0.015	1.9
256	75.9	0.007	1.8	31.3	0.012	3.2
512	80.6	0.008	4.1	36.0	0.007	3.8
1024	80.5	0.010	10.3	35.8	0.012	11.9
2048	80.6	0.014	29.5	35.8	0.012	24.9
4096	81.3	0.015	62.8	36.7	0.014	56.7
8192	82.5	0.017	135.9	37.2	0.015	121.8
16384	89.1	0.013	216.7	44.1	0.014	231.0
32768	93.6	0.011	374.7	48.7	0.012	393.2
65536	104.8	0.011	703.8	58.3	0.011	721.3
131072	128.1	0.010	1355.4	82.6	0.010	1367.4
262144	176.9	0.010	2657.5	130.8	0.010	2664.4
524288	278.5	0.012	6242.9	235.2	0.011	5560.2
1048576	499.2	0.010	10483.9	454.5	0.010	10528.0

By presenting the *bm* quantity, tables 5.1, 5.2 and 5.3 can be used to quickly determine the message size for which the ratio $R = o/bm$ is approximately one. This ratio has an important role for determining the algorithms that can benefit most from early binding, as suggested earlier in Chapter III. Another important observation from the overhead results on the three test configurations is that the overhead grows proportionally with increasing message size. This observation justifies the assumption made in Chapter III that the overhead in realistic systems is a linear function of the message size (*i.e.*, $o(m) = o^{const} + vm$)).

Table 5.3 Measurements of BOUM parameters on *dim-lin-via* configuration

m [byte]	blocking			polling		
	o [μsec]	b [μsec/byte]	bm [μsec]	o [μsec]	b [μsec/byte]	bm [μsec]
0	38.0	n/a	0.0	18.0	n/a	0
4	38.0	0.014	0.1	18.1	0.014	0.1
8	38.0	0.014	0.1	18.0	0.014	0.1
16	38.0	0.014	0.2	18.0	0.014	0.2
32	38.0	0.014	0.5	18.0	0.014	0.5
64	38.0	0.014	0.9	18.0	0.014	0.9
128	38.0	0.014	1.8	18.0	0.015	1.8
256	38.0	0.016	4.1	18.0	0.015	4.0
512	40.4	0.018	9.1	20.4	0.017	8.9
1024	40.4	0.018	18.5	20.4	0.017	17.6
2048	40.5	0.019	38.8	20.4	0.018	37.0
4096	41.0	0.019	78.5	21.0	0.018	74.8
8192	41.5	0.019	154.8	21.5	0.019	154.2
16384	47.0	0.019	314.5	27.1	0.019	311.1
32768	53.7	0.018	585.4	34.1	0.018	582.0
65536	66.5	0.018	1156.1	46.6	0.017	1139.0
131072	92.4	0.017	2241.7	72.6	0.017	2240.4
262144	144.3	0.017	4459.0	128.3	0.017	4464.4
524288	248.5	0.017	8837.9	228.0	0.017	8855.2
1048576	455.5	0.017	17681.4	435.5	0.017	17676.5

5.5 Experimental Results for Early Binding

The experimental results presented in this section are based on the Jacobi and CG iterative solvers described in Chapter III. The results for both algorithms are provided in parallel for each stage of the specified experimental methodology. Since the communication and computation structures of both algorithms are similar, the discussions provided in this section often views the two jointly. Whenever necessary, specifics about the algorithms are outlined.

5.5.1 Implementation of Parallel Algorithms

The parallel implementations of both the Jacobi and CG iterative solvers for systems of linear equations use 1-D data decomposition. The algorithms are based on an iterative procedure for obtaining a vector x (the unknowns in the system) with acceptable precision. This is accomplished through iterating over successive approximations of this vector. In each iteration i ($0 < i \leq k$), the processes in the parallel algorithm compute a local instance of the vector $x^{(i)}$ and exchange this instance with the rest of the processes. At the end of the iteration, a convergence check is performed. The communication is implemented with *MPI_Gather*, *MPI_Bcast*, and *MPI_Allreduce*. The *MPI_Allreduce* operation is used in the convergence check. The *MPI_Gather* and *MPI_Bcast* calls are used for exchanging the local portions of the vector x. The algorithms were executed with problem sizes n = 1,024, 2,048, and 4,096. Both algorithms were executed with precision $\varepsilon = 10^{-5}$. The algorithms were implemented in C and compiled with egcs-gcc 2.92 on Linux or Visual Studio C++ 6.0 on Windows. MPI/Pro was used for message passing.

5.5.2 Estimating Parallel Performance with BOUM

The theoretical expressions for the execution time of both the sequential and parallel implementations of the Jacobi and CG solvers were presented in Chapter III. The asymptotic complexity of the sequential algorithms is defined as $T_s = \Theta(f(n))$. For both of the reviewed algorithms $f(n) = kn^2t_c$, where n is the problem size (*i.e.*, number of equations in the system), k is the number of iterations necessary for convergence, and t_c is the time for a basic unit computation. The number of iterations for convergence is an

important factor in the performance analysis of the iterative solvers. The experiments show that for the selected values of the initial vector $x^{(0)}$, the coefficient matrix A, and the solution vector b, that for all problem sizes the CG algorithm converges for 10 iterations. For the same values of $x^{(0)}$, A, and b, the Jacobi algorithm converges for 22, 21, and 19 iterations for the 1,024, 2,048, and 4,096 problem sizes, respectively.

As indicated earlier, the basic unit computation time is measured as $t_c = T_s/f(n)$, where T_s is the execution time of the sequential implementation and $f(n)$ is the function that represents the asymptotic complexity of the sequential algorithm: $f(n) = kn^2t_c$. The measurements of the sequential times T_s of the two algorithms for all message sizes are shown in Table 5.4. This table also presents the results of t_c for the two algorithms on the *sag-lin-via* cluster configuration. An average value of t_c = 43.2 nanoseconds is used for all further estimations in the this section.

Table 5.4 Computing Jacobi and CG basic unit computaiton time on *sag-lin-via*

n	n^2	Jacobi			CG		
		k	T_s [sec]	t_c [nanosec]	k	T_s [sec]	t_c [nanosec]
1024	1048576	22	0.974	42.2	10	0.453	43.2
2048	4194304	21	3.726	42.3	10	1.811	43.2
4096	16777216	19	13.46	42.2	10	7.225	43.1

Using the value of t_c as computed in Table 5.4, as well as the overhead and bandwidth parameters as measured earlier in this chapter, the parallel performance of the two algorithms can be estimated with BOUM based on the theoretical derivations

presented in Chapter III. Table 5.5 compares the execution times of the Jacobi algorithm as estimated with BOUM and the experimental execution times (with shaded headings). The estimated times are on average within 8% of the measured times, which indicates that BOUM provides an accurate representation of parallel performance for the combination of algorithms and target platform configurations studied.

Table 5.5 Comparison of measured and estimated Jacobi execution times in seconds

size [byte]	1	2		4		8	
	measured	estimated	measured	estimated	measured	estimated	measured
1024	0.974	0.503	0.498	0.261	0.264	0.141	0.130
2048	3.726	1.909	1.880	0.965	0.960	0.532	0.583
4096	13.46	6.894	6.596	3.459	3.406	1.751	1.738

5.5.3 Measuring Degree of Persistence

The degree of persistence d_p was defined in Chapter III as a metric intended to identify the capability of a parallel system to benefit applications that use early binding. Degree of persistence reflects the relative weight of the setup overhead in the total message overhead, $d_p = o_s/o$, where $o = o_s + o_i$. The procedures for measuring the setup and initiation overheads were outlined earlier in figures 5.1 and 5.2. Following these procedures, Table 5.6 summarizes the values of the degree of persistence on the *sag-win-via*, *sag-lin-via*, and *dim-lin-via* test configurations for message sizes in the range [0, 1MB], using MPI/Pro in both blocking and polling modes of notification.

Table 5.6 Measurements of the degree of persistence

m [byte]	sag-win-via		sag-lin-via		dim-lin-via	
	blocking	polling	blocking	polling	Blocking	polling
0	0.14	0.30	0.10	0.23	0.08	0.17
4	0.15	0.32	0.10	0.25	0.08	0.17
8	0.15	0.32	0.10	0.23	0.08	0.17
16	0.15	0.32	0.10	0.25	0.08	0.17
32	0.14	0.33	0.10	0.23	0.08	0.17
64	0.14	0.32	0.10	0.23	0.08	0.17
128	0.15	0.32	0.10	0.23	0.08	0.17
256	0.14	0.32	0.10	0.23	0.08	0.17
512	0.31	0.57	0.15	0.33	0.14	0.27
1024	0.34	0.56	0.15	0.33	0.14	0.27
2048	0.31	0.56	0.15	0.33	0.14	0.27
4096	0.32	0.57	0.16	0.35	0.15	0.29
8192	0.41	0.66	0.17	0.35	0.16	0.31
16384	0.42	0.68	0.23	0.46	0.26	0.45
32768	0.48	0.71	0.27	0.51	0.35	0.56
65536	0.53	0.76	0.35	0.59	0.48	0.68
131072	0.61	0.82	0.46	0.71	0.62	0.79
262144	0.73	0.88	0.61	0.82	0.76	0.88
524288	0.83	0.93	0.75	0.90	0.86	0.93
1048576	0.91	0.97	0.86	0.95	0.92	0.97

The results in Table 5.6 show that the degree of persistence has low values for short messages and high values for long messages, on all tests configurations. The reason for this behavior is explained by fact that the initiation overhead component of the total overhead stays constant for all message sizes, and only the setup overhead changes as the messages grow larger. The setup overhead of MPI/Pro using VIA for transport is primarily dependent on pinning user buffers in physical memory. As mentioned in Chapter III and IV, the time for pinning a buffer is linearly proportional to the number of physical pages that the buffer occupies. This observation indicates that the setup overhead is a continuously growing function, and since the initiation overhead is

independent on the message size, the degree of persistence will increase with increasing message size.

A higher degree of persistence generally indicates that early binding can be successfully used for minimizing the overall overhead by amortizing the setup overhead over multiple communication transactions. However, the overall impact of early binding on the communication performance depends, to a large degree, on the relative weight of the overhead in the total transmission time, $T_c = o + bm$. This relative weight is expressed through the ratio $R = o/bm$. For long messages, this ratio will have a low value because the bandwidth component dominates the total transmission time, which results in lower value of R, hence a low performance gain is realized from early binding. As stated in Chapter III, algorithms that benefit most from early binding are those that use messages with sizes that suggest relatively high values of both R and d_p.

Table 5.6 also shows that the degree of persistence reaches higher values in polling mode than it does in blocking mode. By definition the initiation overhead includes the completion notification time, which is based on kernel synchronization objects in blocking mode and presents a substantial portion of the total overhead. The higher values of the initiation overhead reduce the relative weight of the setup overhead in the total overhead and, therefore, the degree of persistence also decreases.

Table 5.6 also demonstrates that the degree of persistence has lower values on configurations with fast processors than it does on configurations with slow processors, especially for short messages. This observation is explained by the strong relationship

between the setup overhead and CPU speed. Since the setup overhead is primarily a result of activities executed by the message-passing middleware and the low-level communication layers, increasing CPU speed leads to a proportional decrease in the time spent on these activities; hence, the setup overhead is reduced. In contrast, the initiation overhead mainly depends on the communication protocols and the network capabilities, which are affected to a lesser degree by CPU speed. Therefore, the initiation overhead stays almost constant with increasing CPU speed. As a result, the relative weight of the setup overhead in the total overhead is decreased on a fast processor, which leads to smaller values of the degree of persistence.

5.5.4 Implementation of Algorithms with Early Binding

This sub-section presents the results from the Jacobi and CG algorithms with early binding. For improved clarity, only the cumulative communication times of these algorithms are presented. All communication times are expressed in milliseconds. Figure 5.4 presents the communication times of the Jacobi algorithm for problem size n = 1,024. The dashed curve represents the non early binding case. The continuous curve reflects the early binding case. Evidently, for this problem size, the algorithm with early binding reduces the cumulative communication times for all process counts, and specifically for p = 8. This observation has an important implication on the scalability of the algorithm. The scalability analysis based on fixed problem size indicates that increasing the number of processes decreases the message sizes that are produced by the parallel algorithm. The relative weight of the overhead in the total message communication time increases as the

size becomes smaller, as indicated in Table 5.2, which represents the overhead and bandwidth-related components of the communication time. Increasing the relative weight of the overhead reduces the effective time for the actual transmission of data, which has been shown to be one of the major causes for scalability limitations in parallel systems. This effect is reported in the literature review in Chapter II regarding research in the area of performance and scalability analysis. The results in Figure 5.4 show that the negative effect of overhead on scalability can be addressed by early binding. This is an important observation that defines early binding as a software mechanism that can provide both performance enhancements and scalability improvements.

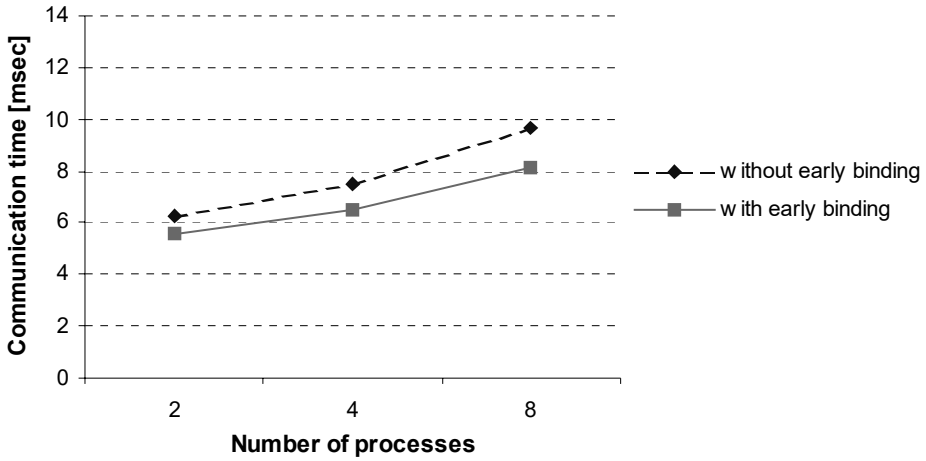


Figure 5.4 Communication times of Jacobi algorithm with problem size 1,024

Figure 5.5 presents the communication times for problem size 2,048 and graphically demonstrates the same behavior as Figure 5.4. The largest performance gain is achieved for p = 8, which indicates that the relative benefit of early binding increases

with scaling the parallel system. Figure 5.6 presents the times for problem size 4,096 and shows the same trend.

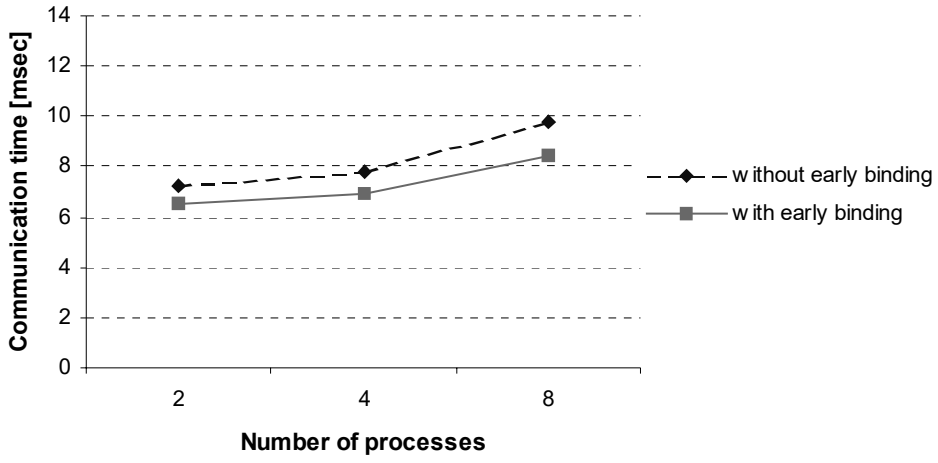


Figure 5.5 Communication times of Jacobi algorithm with problem size 2,048

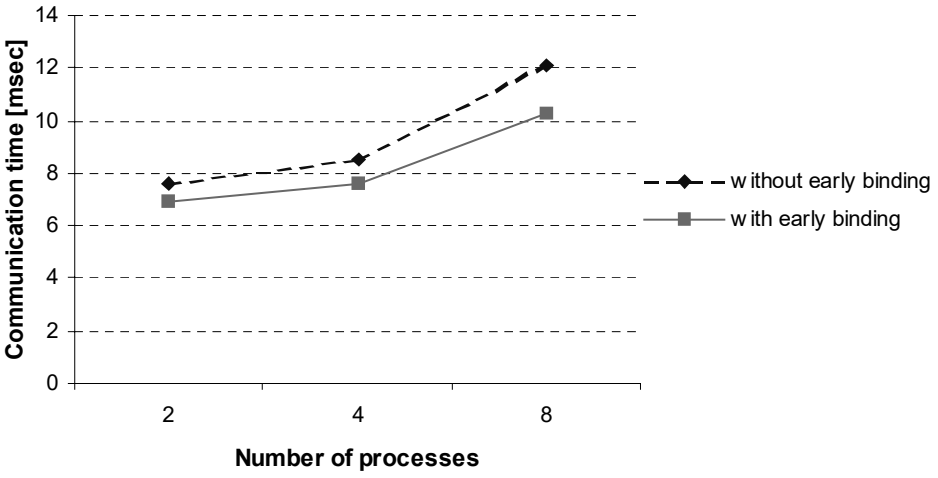


Figure 5.6 Communication times of Jacobi algorithm with problem size 4,096

The results obtained from the CG algorithm reaffirm the observation that early binding provides a significant improvement in communication performance and suggests a mechanism for addressing scalability on large-scale systems. Figure 5.7 presents the cumulative communication times of CG with problem size 2,048.

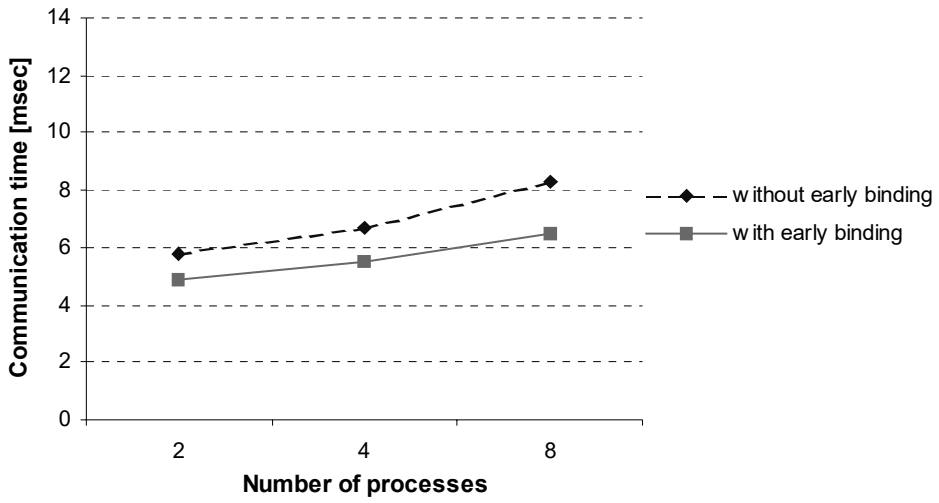


Figure 5.7 Communication times of CG algorithm with problem size 2,048

5.5.5 Validation of Performance Estimates

The performance analysis presented in Chapter III, in the section regarding early binding, provided a general expression for estimating the performance gain when early binding is applied to a specific parallel algorithm, specifically expression (3.16). This expression offers a straightforward method for estimating the communication speedup of early binding S_c, defined as the ratio T_c^{slow}/T_c^{fast}. The estimation is based on the values of the ratio $R = o/bm$ and the degree of persistence d_p for the message size m that is used by the parallel algorithm to exchange the bulk of its data. Table 5.7 provides a breakdown of

the message sizes resulting from each problem size of the Jacobi algorithm for two, four, and eight of processes.

Table 5.7 Message sizes produced by Jacobi algorithm

Problem size	Number of processes		
	2	4	8
1024	2048	1024	512
2048	4096	2048	1024
4096	8192	4096	2048

The estimated communication speedup resulting from early binding is based on the message sizes provided in Table 5.7 and the values of o, b, and d_p as calculated earlier in this chapter. These parameters are substituted in expression (3.36), which yields the estimates for each combination of problem size and number of processes. Finally, Table 5.8 makes a comparison between the performance estimates based on BOUM and the actually measured communication speedups (with shaded column headings).

Table 5.8 Comparison between estimated and measured speedups

problem size	2		4		8	
	estimated	measured	estimated	measured	estimated	measured
1024	1.12	1.13	1.15	1.15	1.17	1.20
2048	1.10	1.11	1.12	1.13	1.15	1.17
4096	1.07	1.10	1.10	1.12	1.12	1.16

As demonstrated in Table 5.8, for most combinations of problem size and number of processes, the performance estimates of the communication speedup are either exactly the same as the measured ones or differ only by a constant factor of 0.02. This shows that the theoretical framework introduced in Chapter III can predict the impact of early

binding on the communication performance of a parallel algorithm with a high degree of accuracy.

5.5.6 Interpretation of Results and Conclusions

The experimental results presented in this section support the hypothesis that early binding is an important source of communication and overall application performance improvement. This was demonstrated by comparing the communication times of two iterative parallel algorithms executed in two modes – with and without early binding. This comparison showed that a 20% cumulative communication time reduction could result from early binding for large number of processes. Another important observation is that the relative performance gain increases with increasing number of processes in a parallel job. This fact provides another justification for the focus on early binding as an important source of performance and scalability.

The experimental results provided in this section support the hypothesis that BOUM, introduced in Chapter III, can accurately describe parallel performance on clusters of workstations. This was demonstrated by comparing the parallel execution times of the Jacobi and CG algorithms with those predicted by BOUM. On average, the difference between actual and predicted performance was within 8%.

Also, this section investigated the capability of the theoretical framework to accurately describe the effect of early binding on performance. This analysis was performed with BOUM and the newly introduced metric degree of persistence. The estimations of the performance gain from early binding were compared to the measured,

actual gains. The estimated gains closely matched the measured ones, which supports the hypothesis that early binding can be described in a theoretical performance model used for accurate prediction of early binding effects. To the best of the author's knowledge, the analysis based on the presented theoretical framework is the first attempt for an in-depth systematic study of early binding in parallel systems.

5.6 Experimental Results for Overlapping

This section presents experimental results and performance analysis for overlapping of communication and computation. The experimental results have been obtained from the parallel FFT algorithm described in Chapter III. This FFT algorithm suggests communication and computation structures that are suitable for overlapping. The performance analysis studies the descriptive power and estimation accuracy of BOUM and also focuses on the factors that determine overlapping efficiency. This section also provides practical procedures for measuring the new performance metrics introduced in Chapter III. These metrics provide valuable information about the capability of a parallel system to support overlapping and also play an important role in the performance analysis of parallel systems.

5.6.1 Implementation of the Parallel Algorithm

The parallel implementation of the studied FFT algorithm uses 1-D data decomposition for distribution of the workload among processes. The algorithm is implemented in C and compiled with egcs-gcc 2.92 on Linux or Visual Studio C++ 6.0

on Windows. The MPI implementation used for message passing is MPI/Pro. The time measurements obtained from the parallel FFT algorithm exclude the bit-reversal procedure for ordering the output elements in the frequency domain. The FFT algorithm with decimation in frequency accepts an ordered sequence in the time domain as input and produces a bit-reversed sequence in the frequency domain. The bit-reversal procedure is considered outside the scope of this study.

5.6.2 Estimating Parallel Performance with BOUM

Chapter III presents a theoretical performance analysis of the parallel FFT algorithm, which emphasizes the effects of overlapping on overall application performance. Using BOUM parameters, expression (3.38) specifies the execution time of the parallel FFT algorithm without overlapping: $T_p = (\log p)T_c(m) + \log n(t_c m_e)$, where $T_c(m) = o + bm$ is the communication time of the messages produced by the algorithm. As indicated earlier, the time estimation for a basic unit computing t_c is performed independently for each combination of algorithm and platform by determining the asymptotic complexity of the sequential algorithm and executing this algorithm on the target platform. The complexity of the sequential FFT algorithm can be expressed as a function f of the problem size n, specifically $f(n) = n\log n$. Then, form (3.36) $t_c = T_s/n\log n$, where T_s is the measured time from the execution of the sequential algorithm. The experimental algorithms were executed with problem sizes n = {512k, 1M, 2M, 4M}. The sequential times for these algorithms and the estimation of t_c for the *sag-lin-via* and *dim-lin-via* configurations are presented in Table 5.9. Since the *dim-lin-via* configuration

is equipped with less memory, only problem sizes 512k and 1M were executed on this configuration.

Table 5.9 Computing FFT basic unit computation time

n	Nlogn	sag-lin-via		dim-lin-via	
		T_s [sec]	t_c [nanosec]	T_s [sec]	t_c [nanosec]
524,288	9,961,472	1.754	176.1	0.987	99.0
1,048,576	20,971,520	3.723	177.5	2.089	99.6
2,097,152	44,040,192	7.903	179.4	n/a	n/a
4,194,304	92,274,688	16.936	183.5	n/a	n/a

After finding t_c, as provided in Table 5.9, all three parameters of BOUM are available, which enables the performance estimation of the FFT algorithm. First, the sizes of the messages exchanged during the global phase of the FFT algorithm are determined. Each FFT element is a complex number with single-precision, floating-point real and imaginary parts; hence, the size of the FFT element is L_e = 8 bytes. Since 1-D data decomposition was used, the work set of each process is n/p elements. During the global phase, each process participates in a pair-wise message exchange with a peer process. The amount of data exchanged is $m = m_e L_e = n/pL_e$. Table 5.10 presents the sizes of the exchanged messages during the global phase of the FFT algorithm as a function of problem size and number of processes in the parallel run.

Table 5.11 presents the measured and estimated execution times on the *sag-lin-via* cluster configuration using MPI/Pro in blocking mode. The results show that the performance estimation of BOUM has a high degree of accuracy. The estimated times differ only about 7% from the actually measured execution times. This fact demonstrates

the descriptive power of BOUM and justifies its selection as a parallel programming model for studying and predicting the performance impact of overlapping and early binding.

Table 5.10 Message sizes of parallel FFT algorithm

p n	2	4	8
512k	2 MB	1 MB	512 kB
1M	4 MB	2 MB	1 MB
2M	8 MB	4 MB	2 MB
4M	16 MB	8 MB	4 MB

Table 5.11 Comparison of estimated and measured execution times in seconds

	1	2		4		8	
size	measured	estimated	measured	estimated	measured	estimated	Measured
1M	3.723	1.963	2.009	1.030	1.068	0.539	0.580
2M	7.903	4.156	4.234	2.174	2.242	1.135	1.231
4M	16.936	8.874	9.088	4.629	4.830	2.411	2.555

5.6.3 Measuring Degree of Overlapping

The measurements for the degree of overlapping are obtained from MPI programs that use the MPI non-blocking, non-persistent API. This API is commonly used for implementing code sections in which communication and computation are overlapped. Thus, the measurements present relevant data to realistic scenarios.

The persistent MPI API suggests a higher degree of overlapping than does the non-blocking, non-persistent API. The persistent API setup overhead can be excluded from the communication time. Since generally overhead cannot be overlapped with computation, eliminating the setup overhead results in a relative increase of the

communication time that can be overlapped. Thus, the persistent API is not only a major facilitator of early binding; it is also an important factor for achieving high degree of overlapping. However, the use of the persistent API depends on specific features of the algorithm and may not be applicable to a range of algorithms that can still benefit from overlapping. Also, the measurements of the degree of overlapping with the persistent API could prove too optimistic for the general case. For these reasons, and in order to obtain results applicable to wider range of algorithms, the MPI test programs used for measuring the degree of overlapping, as well as the CPU overhead and degree of asynchrony metrics, are implemented with the non-blocking, non-persistent API instead of with the more specific persistent API. (This suggests that algorithms and applications willing to make code changes in order to exploit persistence will do even better, in real situations, than the results presented here suggest.)

Table 5.12 presents test results for measuring the degree of overlapping on the *sag-win-via*, *sag-lin-via*, and *dim-lin-via* experimental configurations. As can be seen from this table, the degree of overlapping for all configurations reaches values close to one. This means that all parallel systems represented can support effective overlapping of communication and computation, especially for the message sizes that yield degree of overlapping higher than 0.5. It should be noted that for message size less than or equal to 4,096 bytes, the degree of overlapping is effectively zero. Hence, for these sizes, the benefit of overlapping will be negligible. This behavior of the degree of overlapping is explained by the protocol implementations of MPI/Pro.

Table 5.12 Degree of overlapping

size	sag-win-via	sag-lin-via	dim-lin-via
0	0.00	0.00	0.00
4	0.00	0.00	0.00
8	0.00	0.00	0.00
16	0.00	0.00	0.00
32	0.05	0.02	0.02
64	0.01	0.03	0.03
128	0.02	0.03	0.03
256	0.02	0.04	0.04
512	0.03	0.05	0.03
1024	0.05	0.06	0.04
2048	0.02	0.03	0.02
4096	0.02	0.02	0.01
8192	0.21	0.73	0.67
16384	0.28	0.81	0.79
32768	0.31	0.85	0.86
65536	0.58	0.88	0.90
131072	0.68	0.88	0.92
262144	0.73	0.88	0.93
524288	0.85	0.89	0.94
1048576	0.86	0.89	0.94

As mentioned in Chapter IV, MPI/Pro provides two different message passing protocols: short and long. The main goal of the short protocol is to minimize latency of short messages. This is achieved by avoiding message-passing overhead at the expense of increased CPU usage. Instead of pinning the user buffers for short messages, MPI/Pro makes one copy at each communicating side. These copies are intended to avoid the high-overhead memory pinning operation, which is more time consuming for short message sizes than is the entire “wire” time. The long protocol implements a three-step rendezvous procedure and its main goal is to achieve maximum sustainable bandwidth. In contrast to the short protocol, the long protocol pins user buffers in physical memory at

both the sender and receiver process and uses RDMA operations, thus avoiding the extra copies. For long messages, the additional copies become a major bandwidth-limiting factor. MPI/Pro switches from short to long protocol when the time spent on pinning user buffers becomes shorter than the time spent on memory copies. Typically the switchover size is the size for which the ratio $R = o/bm$ becomes one.

Evidently, the short protocol does not provide sufficient opportunities for overlapping because the CPU is engaged in copying memory of user buffers into previously pinned system buffers. During memory copy, the CPU cannot perform useful computation, which results in a low degree of overlapping. This is also demonstrated in Table 5.13, which shows the CPU overhead for different size messages. For message sizes served by the short protocol, the CPU is actively involved in communication. As soon as MPI/Pro switches to the long protocol, the degree of overlapping increases significantly. If the switchover is shifted toward shorter message sizes, the degree of overlapping for these messages will increase. However, this increase is likely to result in an increased overall communication time because of increased overhead, which can effectively outweigh the benefit of overlapping.

5.6.4 Measuring CPU Overhead

The CPU-overhead metric demonstrates what portion of its cycles the CPU spends on communication. CPU overhead is time that cannot be used for computation. The CPU overhead metric presents another look at the capability of a parallel system to support effective overlapping. If CPU overhead is high, then the parallel system will keep

the CPU busy performing communication-related activities and achieving effective overlapping will be impossible. In contrast, if the CPU overhead is low, the CPU will spend only a small portion of its time on communication and, therefore, more CPU cycles will be dedicated to useful computation. Table 5.13 presents the measurements of the CPU overhead metric for the *sag-win-via*, *sag-lin-via*, and *dim-lin-via* configurations in blocking mode of the MPI/Pro library.

Table 5.13 Processor overhead

size	sag-win-via	sag-lin-via	dim-lin-via
0	0.94	0.65	0.55
4	0.93	0.64	0.54
8	0.92	0.63	0.52
16	0.90	0.61	0.51
32	0.88	0.58	0.48
64	0.85	0.57	0.47
128	0.82	0.53	0.44
256	0.79	0.51	0.49
512	0.77	0.50	0.51
1024	0.74	0.51	0.52
2048	0.70	0.55	0.53
4096	0.65	0.58	0.54
8192	0.49	0.25	0.20
16384	0.40	0.20	0.14
32768	0.22	0.17	0.90
65536	0.15	0.10	0.04
131072	0.11	0.05	0.02
262144	0.09	0.03	0.02
524288	0.07	0.02	0.01
1048576	0.05	0.02	0.01

It is important to note that low CPU overhead alone does not guarantee that the parallel system will provide a high degree of overlapping. Chapter III specifies a number of requirements for effective overlapping; CPU overhead is only one of these

requirements. Sufficient memory bandwidth and asynchronous (independent) message progress are among the other important requirements. Sufficient memory bandwidth is necessary to enable the CPU to access memory locations needed for computation while communication is taking place. If the memory bandwidth is insufficient, even when the CPU overhead is low, the parallel algorithms will exhibit only limited benefits from overlapping. The effect of message progress is demonstrated further in this section.

5.6.5 Measuring Degree of Asynchrony

Independent message progress facilitates overlapping of communication and computation by releasing the main user thread from participating explicitly in activities related to progressing submitted message requests. This enables the user thread to submit a communication request and then schedule a computing activity, which, if the other requirements for overlapping are met, can be overlapped with the communication activity associated with the submitted request. The capability of the parallel system, and especially of the message-passing middleware, to support independent progress is measured by the metric – "degree of asynchrony." This metric was introduced in Chapter III. Degree of asynchrony measures the capability of a parallel system to make asynchronous progress of communication requests while the CPU performs computation.

The pseudo code of the test program for measuring the degree of asynchrony is presented in Figure 5.8. The goal of this code is to measure how much of the communication has progressed during the computation activity. For this purpose, the duration of the communication activity is first measured independently as t_{comm}. Then, the

duration of the communication activity executed concurrently with the computation activity t_{async} is measured as the sum of two components. The first component is the time spent in posting the asynchronous request by **irecv** while the second component is the time spent on synchronization and completion notification in **test/wait**. The degree of asynchrony is determined as $d_a = 1 - t_{async}/t_{comm}$.

If the entire communication activity has completed between the **irecv** and **test/wait** operations, then $t_{async} = 0$ and the degree of asynchrony $d_a = 1$. Inversely, if no communication has taken place between **irecv** and **test/wait**, $t_{async} = t_{comm}$ and $d_a = 0$. The communication time includes the time for synchronization and completion notification. These operations are performed in the context of the user thread; therefore, they cannot be "progressed" independently by the message-passing middleware This fact reduces the effective value of the degree of asynchrony for short messages, as can be seen from Table 5.14 in the columns with shaded headings.

In the MPI test program for measuring degree of asynchrony, **irecv**, **test**, and **wait** are implemented with *MPI_Irecv*, *MPI_Test*, and *MPI_Wait*, respectively. The computation activity is chosen to be significantly longer than the communication activity (*i.e.*, $t_{comp} >> t_{comm}$). It is important to note that the goal of this experiment is not to measure how long the combined execution of the communication and computation activity is. Rather, this experiment determines what portion of the communication can progress independently of the user thread while this thread performs a long computation. The former goal is achieved through the degree-of-overlapping metric presented earlier.

```
tb = time()
Compute();
tcomp = time()- tb

if(rank == 0)
// measure tcomm
      barrier
tb = time()
      irecv(<args>, size = m, src = 1, rreq)
      wait(rreq)
tcomm = time() - tb
// measure tasync
barrier
tb = time()
      irecv(<args>, size = m, src = 1, rreq)
tasync += time() - tb
Compute();
tb = time()
      test(rreq, &flag)
if(flag is false)
wait(rreq)
tasync += time() - tb
else if(rank == 1)
      barrier
      send(<args>, size = m, dst = 0)
barrier
      send(<args>, size = m, dst = 0)
endif
da = 1 - tasync/tcomm
```

Figure 5.8 Pseudo code for measuring degree of asynchrony

Table 5.14 presents experimental measurements for the degree of asynchrony. In order to demonstrate the impact of the MPI message progress capabilities, Table 5.14 compares results obtained with both blocking and polling modes of MPI/Pro. The blocking mode uses asynchronous message completion and independent progress for all message sizes, while the polling mode uses polling notification and polling progress for short messages, and polling notification and independent progress for long messages. Chapter IV presented a detailed description of these modes. Table 5.14 clearly shows that the message progress plays an important role in achieving a high degree of asynchrony.

Table 5.14 Degree of asynchrony

size	sag-win-via		sag-lin-via		dim-lin-via	
	blocking	polling	blocking	polling	blocking	polling
0	0.12	-4.93	0.43	-15.62	0.06	-15.13
4	0.15	-5.02	0.44	-30.31	0.09	-25.19
8	0.17	-4.95	0.43	-30.31	0.15	-25.45
16	0.14	-4.98	0.47	-31.65	0.22	-25.21
32	0.16	-4.96	0.45	-31.72	0.26	-23.20
64	0.18	-4.84	0.46	-30.30	0.32	-27.81
128	0.19	-4.84	0.45	-26.86	0.35	-20.40
256	0.20	-4.74	0.44	-29.23	0.38	-20.25
512	0.20	-4.54	0.49	-23.43	0.44	-0.17
1024	0.23	-4.03	0.49	-16.43	0.59	0.07
2048	0.25	-0.59	0.53	-10.86	0.75	0.42
4096	0.49	0.24	0.68	0.34	0.91	0.70
8192	0.64	0.62	0.76	0.68	0.89	0.81
16384	0.69	0.60	0.81	0.75	0.93	0.89
32768	0.77	0.74	0.87	0.84	0.95	0.93
65536	0.85	0.83	0.92	0.90	0.97	0.95
131072	0.91	0.89	0.94	0.94	0.97	0.97
262144	0.93	0.93	0.95	0.95	0.97	0.97
524288	0.95	0.95	0.96	0.96	0.98	0.97
1048576	0.96	0.97	0.97	0.97	0.98	0.98

Table 5.14 shows that independent message progress (blocking mode) facilitates a high degree of asynchrony for medium- and long-size messages. This means that if the user thread submits a non-blocking communication request and then initiates a computation operation, MPI/Pro will guarantee the timely progress of the message associated with this request. This progress is a critical requirement for effective overlapping. The low degree-of-overlapping values for short messages reflect the fact that the completion notification time for these messages is longer than the actual transmission time. Since completion notification is performed within the context of the

user thread, there is not much to be "progressed" independently of the user thread; hence, the effective value of the degree of asynchrony is low.

In contrast to independent progress, polling progress not only fails to facilitate a high degree of asynchrony, it results in a communication slowdown. This is demonstrated by the negative degree-of-overlapping values for messages that use polling progress (size < 4,096). As specified earlier, the expression for computing degree of asynchrony is $d_a = 1 - t_{async}/t_{comm}$. This expression can produce a negative result only if t_{async} is larger than t_{comm}. If $t_{async} > t_{comm}$, this indicates that attempting to make asynchronous progress in polling mode will yield a performance slowdown. The fact that t_{async} becomes larger than t_{comm} is explained by the fact that for a receive operation, the polling progress engine selects some constant number of iterations to execute aggressive polling. This polling checks whether a matching message should arrive shortly after the receive request is posted (denoted with **irecv** in Figure 5.4). The number of aggressive polls is implementation-dependent and is often determined empirically. However, its success depends, to a large degree, on the behavior of the user algorithm as well as the underlying system hardware and software. If the aggressive polling does not match an incoming request, then the time spent on polling is pure overhead that results in an effective increase of the communication time.

This observation has critical importance for validating the hypothesis that although polling progress, together with polling notification, can yield the lowest latency as measured by ping-pong tests, polling progress does not ensure timely transmission of

asynchronous messages, which can result in an effective communication slowdown. This scenario was described in Chapter IV, Figure 4.1 where the impact of message progress on the effective bandwidth was studied. The experimental results obtained from the tests for measuring the degree of asynchrony prove the hypothesis that polling progress results in reduced effective bandwidth in real parallel algorithms with non-trivial communication and computation structures. The behavior of algorithms with such structures is not captured adequately by the ping-pong latency test; hence, ping-pong tests cannot provide realistic estimations of the overall capabilities of a parallel system to achieve optimal application performance. The ping-pong tests represent a narrow range of traffic patterns rarely seen in realistic algorithms.

The results for the degree-of-asynchrony metric support the hypothesis that the ping-pong test provides only a limited view on the performance of a parallel system, and more elaborate evaluation procedures and different performance analysis approaches are necessary. This chapter provides such analysis, which is based on a number of new performance metrics, specifically: degree of overlapping, segmentation efficiency, degree of asynchrony, degree of persistence, and also on the commonly used CPU overhead metric.

5.6.6 Results from Optimized FFT Algorithm with Overlapping

This subsection presents experimental results obtained from the parallel FFT algorithm implemented with overlapping. The selection of the number of overlapped segments s is implemented as a run-time option, which facilitates the collection of

voluminous experimental data. Experimental results only from the global phase of the FFT algorithm are presented. As mentioned in Chapter III, the parallel implementation of FFT presented in this work suggests a two-phase structure: a global phase that includes communication and computation, and a local phase that includes only computation. Since the goal of overlapping is to effectively hide communication, this performance-enhancing technique can be applied only to segments of the algorithms that both communicate and compute. Therefore, the scope of this section is focused on the global phase.

Table 5.15 presents the execution times of the global phase of the FFT algorithm on the *sag-lin-via* configuration for three problem sizes: 1M, 2M, and 4M, as specified in the first row of the table. The second row represents the number of processes working on a particular problem size. The first column of the table specifies the number of segments used for the implementation of algorithm's overlapped portion. All experimental results are performed using MPI/Pro in the blocking mode. If the completion notification mode of MPI/Pro is not specified in the following discussion, blocking mode should be assumed. Later in this section, specific analysis of the impact of MPI/Pro's completion notification mode on the effective overlapping is presented. In this analysis, results obtained in both modes are explicitly compared.

The results from Table 5.15 clearly show that overlapping of communication and computation significantly improves the overall performance of the global phase of the FFT algorithm. Improvements of up to 30% over the non-optimized algorithm with overlapping are achieved. Also, this table shows that the number of segments used for

overlapping plays an important role in reaching optimal performance. In order to illustrate the dependency of the benefit of overlapping on the number of segments, the graphs in figures 5.9, 5.10, and 5.11 depict the change of execution time as a function of problem size, number of segments, and number of processes.

Table 5.15 Execution times for FFT global phase

s	1M			2M			4M		
	2	4	8	2	4	8	2	4	8
1	0.212	0.207	0.180	0.425	0.417	0.343	0.855	0.833	0.675
2	0.176	0.173	0.144	0.354	0.348	0.288	0.714	0.696	0.574
4	0.160	0.157	0.133	0.345	0.315	0.269	0.645	0.627	0.524
8	0.152	0.158	0.134	0.327	0.316	0.258	0.627	0.627	0.514
16	0.150	0.160	0.152	0.301	0.316	0.264	0.624	0.629	0.517
32	0.150	0.167	0.146	0.298	0.324	0.279	0.605	0.628	0.545
64	0.154	0.179	0.181	0.320	0.334	0.297	0.598	0.635	0.545

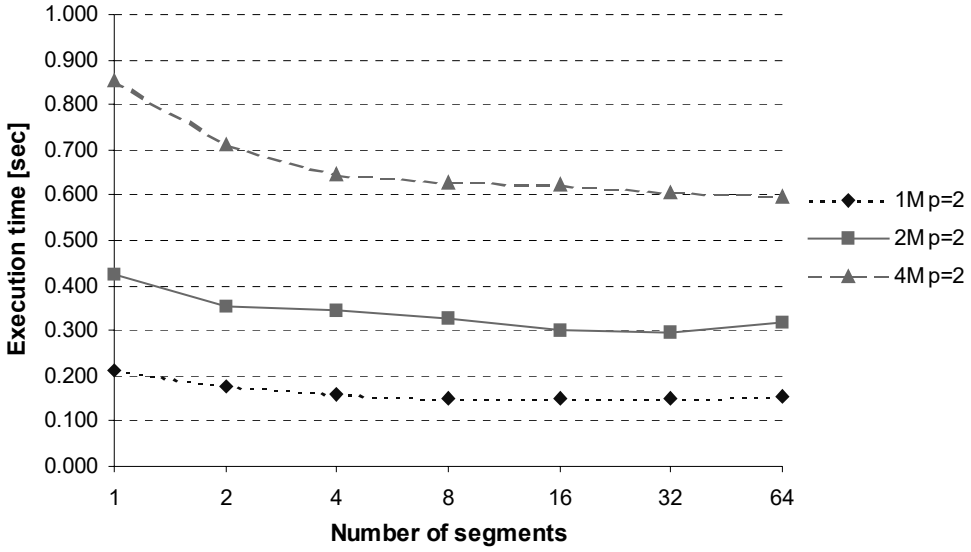


Figure 5.9 Overlapped execution time on *sag-lin-via* with $p = 2$

The graph in Figure 5.9 demonstrates that the effect of overlapping increases with increasing the number of segments. However, this effect has non-linear behavior as predicted by the analysis in Chapter III. The effect quickly reaches optimal values for *s* in the range [4, 8] and subsequent relative gains are much smaller. This behavior is attributed to the fact that, by breaking the exchanged messages into segments, the relative value of the message overhead grows, which affects the value of the degree of overlapping, as reported in Table 5.12. As predicted by the analysis in Chapter III, the benefit of increasing the number of segments is affected negatively by the increased cumulative overhead of the smaller segments. The general shape of the curves in Figure 5.9, as well as in figures 5.10 and 5.11 follows the shape predicted earlier by the theoretical analysis of overlapping as a function of number of segments and depicted in Figure 3.10.

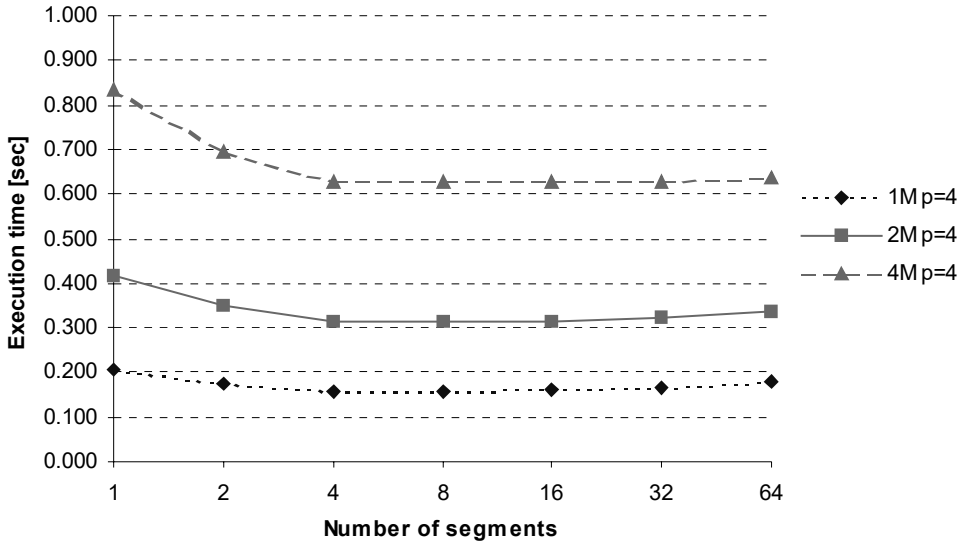


Figure 5.10 Overlapped execution time on *sag-lin-via* with $p = 4$

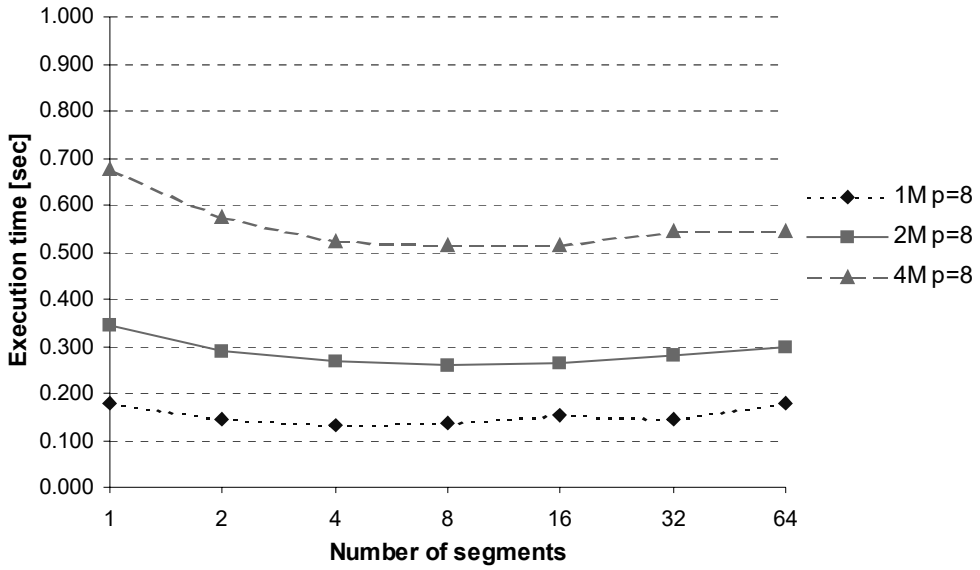


Figure 5.11 Overlapped execution time on *sag-lin-via* with $p = 8$

An interesting trend can be noticed when the graphs in Figures 5.9, 5.10, and 5.11 are compared. The curves on the graph with $p = 2$ reach their optimum point for number of segments ≥ 32. In contrast, the curves on the graph that represents process counts $p = 8$ reach their optimal point for number of segments $s \approx 8$. This difference is explained by the difference in message sizes exchanged in the particular runs. The message sizes for $p = 2$ are four times larger than the message sizes for $p = 8$. This trend shows once again the impact of the overhead on overlapping. For shorter messages (*i.e.*, larger p), the relative weight of the cumulative overhead for the same number of segments becomes larger because the overhead has higher relative weight in the transmission time of shorter segments. In order to illustrate this dependency, the curves of one problem size for the three different process counts are presented in Figure 5.12.

Figure 5.12 shows that the optimal point s_{b2} for the curve that corresponds to p = 2 is between s = 16 and s = 32, the optimal point s_{b4} for p = 4 is between s = 8 and s = 16, and the optimal point s_{b8} for p = 8 is at $s \approx 8$. Therefore, it can be concluded that $s_{b8} < s_{b4} < s_{b2}$. This demonstrates that reducing message sizes (either by increasing the number of processes or by reducing the problem size) shifts the optimal number of segments to lower values and also decreases the relative impact of overlapping.

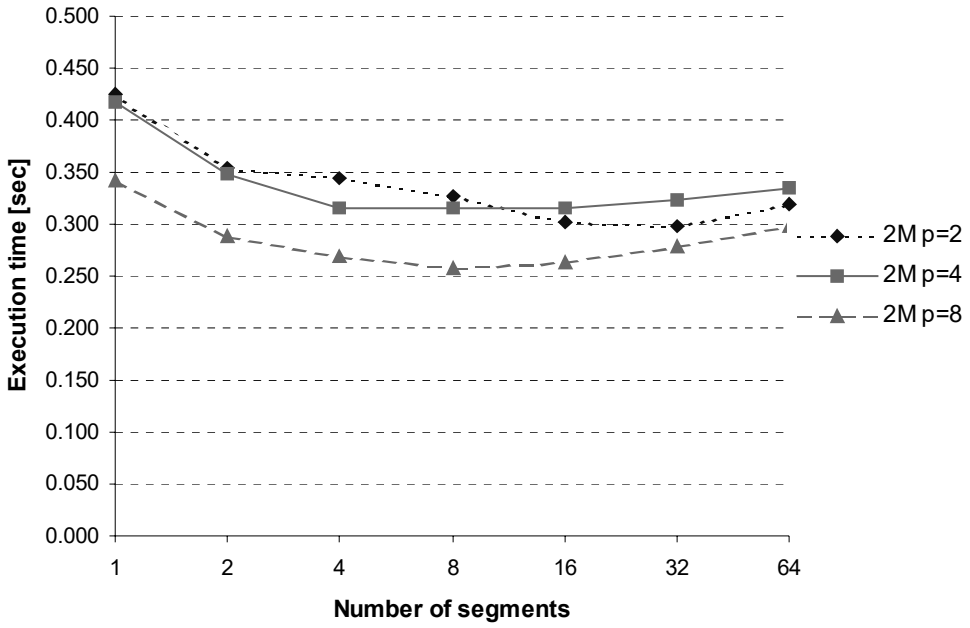


Figure 5.12 Overlapped execution time of 2M FFT on *sag-lin-via* with varying *p*

5.6.7 Impact of MPI Completion Notification on Overlapping

This subsection focuses on the impact of the completion notification mechanism of the MPI library on the effect of overlapping. In the hypothesis of this dissertation, it

was asserted that message-passing libraries with asynchronous (blocking) completion notification provide more opportunities for effective overlapping than do libraries with synchronous (polling) notification. In order to verify this hypothesis, results from the same problem sizes on the same configurations but with different completion notification modes (of MPI/Pro) are presented in Figures 5.13 and 5.14. Figure 5.13 compares the effect of overlapping on execution time for problem size n = 2M and p = 2. Clearly, the graph shows that although the curve shape of the two modes is similar, the absolute gain of blocking mode is significantly higher than that of polling mode. Both curves reach their optimum for s in the same range [16,32].

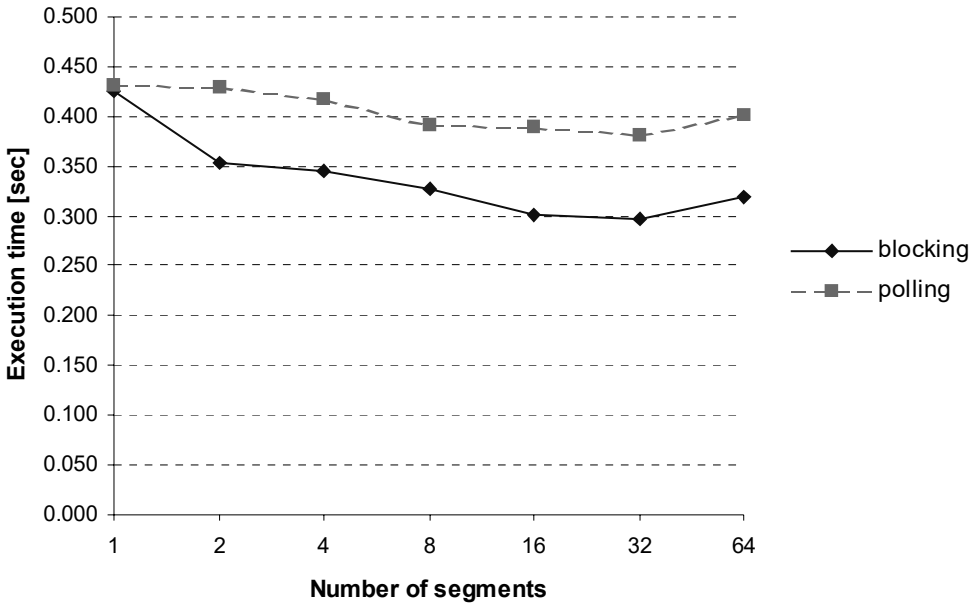


Figure 5.13 Impact of completion notification on overlapping for n = 2M and p = 2

However, the best execution time for polling mode is only 11% better than that of the non-overlapped implementation, while the best execution time for blocking mode is 30% better than that of the non-overlapped time. This comparison clearly shows the superiority of blocking mode over polling mode with respect to supporting effective overlapping. This conclusion is further supported by Figure 5.14 that presents the execution times of the same problems size for p = 8. In this figure, both the blocking and polling mode curves have almost the same shape and both curves reach their optimum for approximately eight segments. However, as demonstrated earlier for p = 2, the performance gain of polling mode is much smaller than the performance gain of blocking mode. In fact, for problem size n = 2M and number of processes p = 8, the maximum gain of polling mode is only 6% while the maximum gain in blocking mode is 25%.

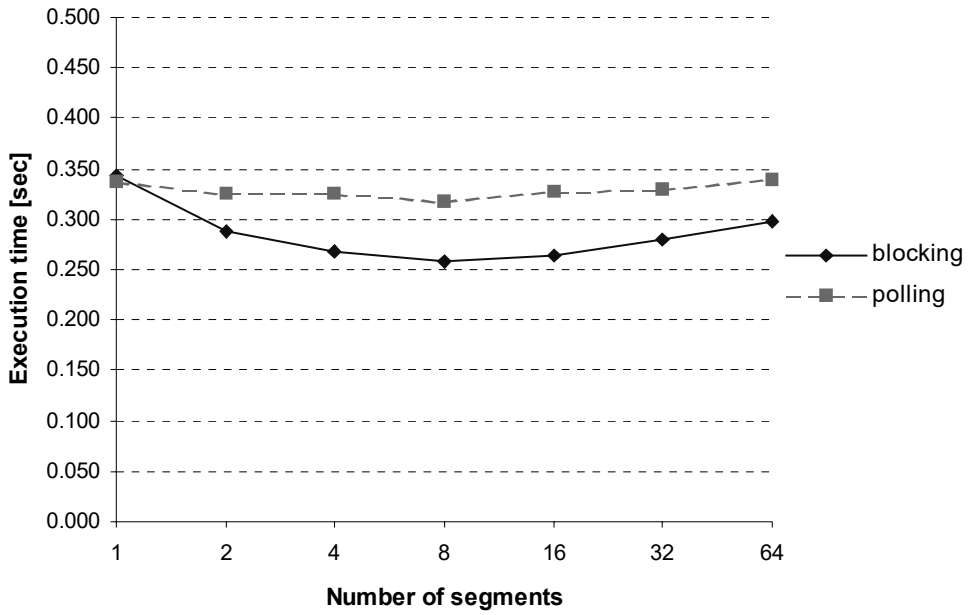


Figure 5.14 Impact of completion notification on overlapping for n = 2M and p = 8

5.6.8 Comparison between Experiments and Theoretical Analysis

The optimal number of segments s_b was theoretically derived in expression (3.51). Using this expression and the values of d_o and R measured earlier, the estimated optimal number of segments for the parallel FFT with problem size n = 2M and number of processes p = 2, 4, and 8 are respectively 13, 12 and 10. The experimental results for this problem size were presented in Figure 5.12. From this figure, it can be assessed that the optimum number of segments for the execution time for p = 2 is in the range [16, 32], for p = 4 is in the range [8, 16], and for p = 8 is approximately equal to eight. Although the estimated optimal numbers of segments do not exactly match the experimentally measured ones, the estimates can serve as a good indication about the range and the relationship between the optimal points for different problem sizes.

5.6.9 Interpretation of Results and Conclusions

This section presented experimental results from a parallel implementation of FFT optimized using overlapping of communication and computation. The results demonstrated that overlapping is an important source of parallel performance improvement that can be applied to a range of algorithms with suitable communication and computation structure. A common approach for achieving effective overlapping was reviewed, namely identifying communication transactions that can be divided into smaller segments, and subsequently overlapped in a pipelined fashion with independent computation. This section provided experimental results that demonstrated the impact of number of segments on overlapping. In addition, this section validated the descriptive and

predictive capabilities of BOUM by comparing the estimated execution times with the experimental execution times. The estimations based on the model differed from the actual measurements by only a small fraction indicating that BOUM accurately captures the hardware and software specifics of the parallel system as experienced by the applications.

This section also demonstrated the descriptive power of a set of new metrics defined in Chapter III, specifically degree of overlapping and degree of asynchrony. These metrics, together with CPU overhead, provide an in-depth look at the complex interactions between the software and hardware components of a parallel system and how these interactions affect overlapping. Further, this section provided support of the hypothesis that an MPI library with asynchronous completion notification and independent progress facilitates effective overlapping, while MPI libraries with polling notification and polling progress offer only minimal benefits of overlapping. Also, it was shown that software architectures that provide the lowest point-to-point latency as obtained by ping-pong tests exhibit sub-optimal behavior when subjected to such common techniques as asynchronous processing and overlapping. In fact it was shown that polling MPI libraries result in effective communication performance slowdown when asynchronous processing is attempted.

CHAPTER VI

SUMMARY, CONCLUSIONS, AND FUTURE WORK

This dissertation investigated advanced software techniques for improving performance on clusters of workstations and approaches for designing efficient message-passing middleware, specifically MPI middleware. This investigation focused on MPI middleware capabilities needed to deliver maximum performance to user processes and provide efficient support for early binding, asynchronous processing, and overlapping of communication and computation. The findings of this work are relevant to a wide audience inasmuch as cluster computers using MPI for interprocess communication are the preferred choice for upgrading existing or building new scalable computing facilities.

This work presented an in-depth study of the performance-related architectural characteristics of clusters and provided a comparison between clusters and traditional Massively Parallel Processors (multicomputers). This comparison revealed that, although clusters have a number of features similar to multicomputers, they differ significantly in important performance-impacting areas. These differences justify a new look at the overall architecture of parallel systems based on clusters, and specifically, on the design of the communication software stack incorporating the low-level messaging system software and the message-passing middleware.

Clusters interconnected with high-speed networks, such as the VIA-based Giganet, were the main area of concentration of the experiments accomplished in this study. These experiments were compared with results obtained on clusters interconnected with commonly used networks, such as Ethernet with the TCP/IP transport. High-speed networks, and especially the networks that are built according to the VIA specification, represent a current trend in network design to support modes of operation that facilitate traditional communication performance parameters, such as low latency and high bandwidth, and at the same time provide other important features that drive the improvement of overall application performance. Among these features are early binding, low CPU overhead, asynchronous processing, and overlapping of communication and computation

These valuable and widely applicable performance-enhancing techniques are presently underutilized in practice because of inadequate support by existing message-passing libraries, and are rarely considered by parallel algorithm designers. Furthermore, commonly accepted methods for performance analysis omit these techniques and focus primarily on more obvious communication characteristics, such as latency of short messages and bandwidth of longer messages. This dissertation addressed these questions and concentrated on describing early binding and overlapping of communication and computation as fundamental approaches for improving parallel performance and scalability. These approaches can be applied successfully to a number of practical cases and can also be used in conjunction with other, more direct approaches, such as

increasing CPU speed, memory bandwidth, peripheral bus throughput, and network physical data rates.

6.1 Proof of Hypothesis and Summary of Findings

This dissertation proved the hypothesis that early binding and overlapping are important sources of performance improvement. This was demonstrated by practical experiments with common parallel algorithms that were optimized by early binding and overlapping. The experimental methodology used in this work provided a performance comparison of the algorithms before and after applying the optimizations studied. This comparison clearly revealed the performance benefits of early binding and overlapping.

Further, this work proved the hypothesis that theoretical models for parallel computation can effectively describe early binding and overlapping, and also successfully estimate the performance benefits of these mechanisms. A theoretical framework, consisting of the Bandwidth and Overhead-based User-level parallel processing Model (BOUM) and a set of new performance metrics, was introduced to address the insufficient support for early binding and overlapping in existing parallel models. The performance analysis based on this framework provided expressions for parallel performance that explicitly account for early binding and overlapping. The descriptive power of the theoretical framework was demonstrated by experimental validation of the accuracy of BOUM to estimate algorithm's absolute performance and an algorithm's performance gain when optimized using the studied performance-enhancing mechanisms.

This dissertation introduced a set of new performance metrics whose goal is an abstract and accurate representation of parallel performance. These metrics facilitate a quantitative analysis of complex system hardware and software interactions as well as interactions between application and system software. The new metrics are degree of persistence, degree of overlapping, segmentation efficiency, and degree of asynchrony. Also, an analysis of the commonly used CPU overhead characteristic of communication systems and its impact on overlapping was presented. The new metrics capture specifics of parallel systems, and especially of message-passing middleware, which are often ignored by performance analysis based on traditional models and metrics. This work attributed the limited extant use of early binding and overlapping in practice to a lack of theoretical description and relevant metrics. This dissertation addressed this insufficiency by providing a theoretical framework based on a specifically developed model and a set of new metrics. This framework provides explicit description of early binding and overlapping and also predicts the performance impact of these mechanisms.

This dissertation also demonstrated that message-passing middleware plays a critical role in either propagating or masking the performance capabilities of lower communication stack layers to user applications. The architecture of the message-passing middleware and its impact on overall application performance is often studied with insufficient depth. Common performance evaluation procedures are reduced to measuring point-to-point latency of short messages using simple ping-pong tests. These tests have insufficient descriptive power and present a limited view of the parallel system as a

whole, and the message-passing middleware in particular. This view fails to account for such performance-improving sources, such as overlapping of communication and computation, asynchronous processing, and early binding. Furthermore, point-to-point performance metrics provide only one dimension of the communication infrastructure. For both an accurate and relevant analysis, aggregate performance metrics are also necessary. The ping-pong tests put unnecessary weight on short-message latency. A large number of medium- to coarse-grain data parallel algorithms use long messages that exhibit reduced latency influence and increased bandwidth influence. Since the problem size of many applications is scaled up as the number of processors of the parallel system is scaled, the data granularity of these applications tends to be maintained even.

This work demonstrated that message-passing libraries with architectures that facilitate the lowest latency numbers achieve this by sacrificing CPU cycles in order to reduce message overhead. This, in turn, affects the capability of these libraries, and hence, the entire parallel system, to perform asynchronous message progress and to provide effective overlapping of communication and computation.

The author demonstrates the benefits of overlapping, asynchronous processing, and early binding by using practical algorithms implemented with MPI/Pro. While MPI/Pro is a commercial-grade implementation of the MPI-1.2 standard, it has been extensively used in the design and performance evaluation discussions presented here, without being the primary deliverable of this work in and of itself. The major design goal of MPI/Pro was delivering maximum communication performance to parallel

applications without sacrificing the computing resources of the host system. In order to achieve this goal, MPI/Pro provides a number of architectural choices for completion notification and message progress that enable various modes of operation. MPI/Pro offers its users the ability to tailor the behavior of the MPI library to their needs, specifically, they can select the completion notification and message progress modes through a run-time switch. Using this unique feature, the author performed a number of experimental tests for proving the hypothesis of this work. These experiments demonstrated that MPI/Pro facilitates effective use of overlapping and early binding.

This work extensively study approaches for improving parallel performance on clusters of workstations interconnected with high-speed networks. Clusters are commonly regarded as a generic instance of traditional multicomputer parallel architectures. This view offers a quick and low cost transition to the new parallel environment. As a result, well-known practices, software packages, and performance analysis techniques are generally applied to clusters. However, clusters differ from multicomputers in a number of important areas. By identifying these differences (presented in Chapter II), this work justifies the special approach to cluster software design and performance analysis.

Presently, most of the cluster users in the world are building applications on middleware that was initiated as research or proof-of-concept efforts and usually designed to be portable, while also providing portable services. The design of such middleware does not account for important performance-enabling features, such as asynchrony, overlapping, and early binding. Some of the reasons for these design

omissions are based on evidence that the communication infrastructure available at the time of initial design did not provide sufficient support for these features (although early binding has always been optimizable to a certain degree). Certain design choices impose strong limitations on the behavior of user applications. As a result, legacy MPI programs, as well as newer MPI programs, tend to miss the benefits of MPI's performance-revealing semantics. Consequently, even newer MPI implementations may not generate significant gains for such applications, without application modification.

This work provided an in-depth analysis of message-passing middleware architectures and their implications on communication and overall application performance. This analysis revealed the importance of such architectural solutions as the methods for completion notification and message progress. Also, this work provided a new MPI implementation that, in addition to the commonly recognized latency and bandwidth performance features, also facilitates effective use of early binding and overlapping of communication and computation. The author’s wish is that this MPI implementation, together with the suggested new metrics and performance analysis, will attract the attention of message-passing middleware and application software designers and expand the scope of the studied software approaches for optimizing parallel performance.

6.2 New Terms and Concepts

This dissertation considered issues that were insufficiently described before in the theory and practice of parallel processing. Therefore, in order to capture important

performance–related interactions and behaviors of the components of a parallel system, a number of new concepts and terms were introduced. These new terms and concepts are summarized as follows:

- A methodology for classification of message-passing middleware libraries was offered. This classification is based on the methods for message completion notification and message progress. Asynchronous (blocking) and synchronous (polling) methods for completion notification were specified. The message progress was classified as asynchronous (independent) and synchronous (polling). According to the introduced classification methodology, four categories of message-passing libraries were identified. MPICH is an instance of all-polling libraries. MPI/Pro enables users to tailor MPI/Pro's behavior according to this classification. Specifically, MPI/Pro supports a mode with blocking notification and independent progress, an all-polling mode for short messages (similar to MPICH's design), and a mode with polling notification and independent progress for long messages.
- A model for parallel computation (BOUM), based on performance attributes as observed by applications, was introduced. This model is an important component of the presented theoretical framework. The major requirement of the model is to provide an explicit description of early binding and overlapping. The model was used to predict the performance of parallel algorithms. The accuracy of the model was verified through an experimental procedure, presented in Chapter V.

- Degree of persistence. This is a new performance metric that captures the capability of a communication system to support effective early binding. Using this metric, parallel software designers can predict the actual benefit of applying early binding to algorithms. Also, system software designers can use this metric to evaluate the quality of their implementation on a particular hardware platform and operating system.
- Degree of overlapping. The goal of this performance metric is to facilitate quantitative analysis of a parallel system's capacity to deliver maximum effect of overlapping of communication and computation to the application processes. This is an important tool for estimating the performance benefits of overlapping on a particular parallel system. This metric captures a number of system features that are difficult to analyze but significantly affect the efficiency of overlapping. These range from purely hardware features, such as memory bandwidth and peripheral bus throughput, to purely software features, such as the architecture of the message-passing middleware.
- Segmentation efficiency. This metric was introduced in order to assist parallel algorithm designers that employ overlapping of communication and computation. A common approach for implementing overlapping is to break large messages into smaller segments that can be pipelined and overlapped with computation. The effectiveness of the segmentation procedure depends on the ratio between the overhead and bandwidth components of the segment communication time on the

target platform. Segmentation efficiency provides a guideline for determining the optimal number of segments that yields maximum overlapping.

- Degree of asynchrony. This metric was introduced to quantify the capability of a system to move user data while user processes are performing activities unrelated to communication. This capability is an important prerequisite for effective overlapping. The degree-of-asynchrony metric was used to assess the capabilities of different message-passing middleware architectures to support asynchronous message progress. The analysis based on this metric clearly showed that libraries that use polling progress exhibit sub-optimal behavior when user processes attempt to schedule concurrent communication and computation activities.

6.3 Future Work

A number of interesting issues remain for future work at the conclusion of this dissertation. Work is needed to provide fully asynchronous collective communication functions, as shown in FastMPI and MPI/RT. Asynchronous collective communication supports overlapping of communication and communication in a broader sense than is supported by point-to-point asynchronous operations. Since a large number of parallel algorithms use collective communication primitives in their performance critical parts, these applications cannot benefit from overlapping. Some of these problems can be addressed by augmenting MPI's API and services for better persistent communication, as considered in Appendix A.

This study focused on cost-neutral software techniques for improving overall parallel performance. Specifically, the impact of asynchronous processing on short-message latency was investigated. It was assessed that the software infrastructure to support these techniques efficiently results in fixed increase of communication overhead for short messages, which, however, is outweighed by the overall performance gains obtained by using these techniques. This study can be further extended to investigate fixed/variable-cost trade-off approaches in which the initialization cost of a given mechanism may be high, but such a one-time investment may deliver substantially higher performance when reused many times.

This dissertation considered early binding and overlapping of communication and computation as two fundamental sources of parallel performance improvement on clusters. These software mechanisms were studied independently and their effect on performance was demonstrated individually. This work showed that the communication overhead of message segments when overlapping is performed leads to a diminishing effect as the number of segments grows. It was also shown that, for certain classes of algorithms, early binding could significantly reduce the effective cost of overhead. This might be used to offset the negative effect of segmentation on the efficiency of overlapping. A future study may focus on the combined effects of overlapping and early binding and also investigate their interaction with other software approaches for performance improvement.

The MPI implementation provided here was specifically optimized for VIA networks. VIA offers a number of advanced features that facilitate efficient communication with low CPU overhead, asynchronous processing, and overlapping. The VIA standardization forum (VIDF) is currently defining quality-of-service attributes that can be attached to VI connections. Future research could study how these attributes may be used by MPI, or other message-passing middleware systems, such as MPI/RT, for achieving more efficient, predictable communication further improving overall application performance and/or predictability. Also, careful analysis of the VIA communication model suggests that there are untapped opportunities for speeding up the transmission of very short messages. This speedup can be achieved by a specialized transfer scheme for short messages, rather than using the general-purpose descriptor-based scheme, which requires unnecessary transactions on the peripheral bus.

New technologies, such as InfiniBand, VI/TCP, and the integration of storage and networking, will accelerate the need for middleware to be cognizant of resource utilization issues, and to provide emerging options for quality of service, both for message passing and for concurrent and streaming I/O. Furthermore, exploration deeper into the question of direct exploitation of distributed shared memory concepts with InfiniBand (and VI Architecture) remains important complements to the infrastructure offered by the MPI interface.

Future work is needed in order to explore questions such as co-scheduling between user threads, progress threads, and system activities. Clearly, the importance of

threads in concurrent processing is increasing, and hardware thread support will be the norm in just a few years. This trend suggests that more users will encounter issues related to message passing in multithreaded environments. The author, in collaboration with Anthony Skjellum and Matthew Gleeson, has provided an initial study on the issues related to achieving efficient use of multi-grained parallelism with MPI and threads. The results of this thread-oriented study are provided in Appendix B.

REFERENCES

Alexandrov, A., M. F. Ionescu, K. E. Schauser, and C. Sheiman. 1995. LogGP: Incorporating Long Messages into the LogP Model – One Step Closer towards a Realistic Model for Parallel Computation. In *Proceedings of the 7th Symposium on Parallel Algorithms and Architectures*, 95-105.

Amdahl, G. M. 1967. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the AFIPS Conference*, 483-485.

Andersen, R. S., R. P. Brent, and D. J. Kuck. 1976. *The Complexity of Computational Problem Solving*. St. Lucia, Queensland, Australia.

Andersen, Ed, Jeff Brooks, Charles Grassl, and Steve Scott. 1997. Performance of the Cray T3E Multiprocessor. In *Proceedings of the ACM/IEEE SC'97 Conference*, San Jose, CA. ACM Press and IEEE Computer Society Press.

Baden, Scott B. and Stephen J. Fink. 1998. Communication Overlap in Multi-tier Parallel Algorithms. In *Proceedings of the ACM/IEEE SC'98 Conference,* Orlando, FL. ACM Press and IEEE Computer Society Press.

Bailey, D., E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnam. 1991. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications* 5 (3): 63-73.

Barnett, M., S. Gupta, P. Dayne, L. Shuler, and R. van de Geijn. 1994. Interprocessor Collective Communication Library (InterComm). In *Proceedings of the IEEE Supercomputing'94 Conference*. IEEE Computer Society Press.

Basu, A., V. Buch, W. Vogels, T. von Eicken. 1995. U-Net: A User-Level Network Interface for Parallel and Distributed Computing, In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, 1993.

Bilardi, G., Kieran T. Herley, Andrea Pietracaprina, Geppino Pucci, and Paul Spirakis. BSP vs. LogP. 1996. In *Proceedings 8th ACM Symposium on Parallel Algorithms and Architectures*, 1993, 25-32.

Blumrich, Matthias A., Kai Li, Richard Alpert, Cezary Dubnicki, and Edward W. Felten. 1994. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 142-153.

Blumrich, Matthias A., Cezary Dubnicki, Edward W. Felten, and Kai Li. 1996. Protected, User-Level DMA for the SHRIMP Network Interface. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*.

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

Brightwell, Ron and Lance Shuler. 1996. Design and Implementation of MPI on Puma Portals. In *Proceedings of the MPI developers conference*, Notre Dame, Indiana.

Brightwell, Ron and Arthur B. Mccabe. 2000. Scalability limitations of VIA-based technologies in supporting MPI. In *Proceedings of the 4th MPI developer's and user's conference*, Ithaca, NY, March 2000.

Bruck, Jehoshua, Danny Dolev, Ching-Tien Ho, Marcel-Catlin Rou, Ray Strong. 1997. Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations. *Journal of Parallel and Distributed Computing* 40 (1): 19-34

Burden, Richard L. and J. Douglas Faires. 1985. *Numerical analysis, 3rd ed*. Boston, MA: Prindle, Weber & Schmidt Publishers.

Compaq Computer Corporation, Intel Corporation, and Microsoft Corporation. 1997. *Virtual Interface Architecture Specification: Version 1.0*. http://www.viarch.org/html /Spec/document/san_10.pdf (Accessed 1 February 1998).

Compaq Computer Corporation. *Compaq ServerNet*. http://himalaya.compaq.com/ view.asp?PAGE=ServerNet (Accessed 8 March 2001).

Cornell Theory Center. *AC3 Technologies*. http://www.tc.cornell.edu/AC3 /tech/index.html (Accessed 8 March 2001).

Cruz, John and Kihong Park. 1999. Toward Performance-Driven System Support for Distributed Computing in Clustered Environments. *Journal of Parallel and Distributed Computing* 59 (2): 132-154.

Culler, David E., Richard M. Karp, David Patterson, Abhijit Sahay, Eunice E. Santos, Klaus E. Schauser, Ramesh Subramanian, and Thorsten von Eicken. 1996. LogP: A Practical Model of Parallel Computation. *Communications of the ACM* 39(11): 78-85.

Dagum, Leonardo, and Ramesh Menon. 1998. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering* 5(1).

Dimitrov, Rossen and Anthony Skjellum. 2000. Impact of latency on applications' performance. In *Proceedings of the 4th MPI developer's and user's conference,* Ithaca, NY, March 2000.

Dimitrov, Rossen and Anthony Skjellum. 1999. An Efficient MPI Implementation for Virtual Interface Architecture – Enabled Cluster Computing. In *Proceedings of the 3rd MPI developer's and user's conference*, Atlanta, GA, March 1999 15-24.

Dimitrov, Rossen, Anthony Skjellum, and Boris Protopopov. How data transfer modes and synchronization schemes affect the performance of a communication system based on Myrinet. Technical Report #970318, Department of Computer Science, Mississippi State University, March 1997

Dimitrov, Rossen and Matthew Gleeson. 1998. Challenges and new technologies for security in high-performance distributed environments. In *Proceedings of the 21st National information systems security conference*, Arlington, VA, (2): 457-468.

Dimitrov, Rossen. 1997. *A Windows NT kernel-mode device driver for PCI Myrinet LANai 4.x interface adapters*. M.S. Project, Mississippi State University.

Dolphin Interconnect. *The scalable coherent interface (SCI)*. http://www.dolphinics.com/scitech.html (Accessed 8 March 2001).

Flatt, H. P. and K. Kennedy. 1989. Performance of Parallel Processors. *Parallel Computing* 12: 1-20.

Fortune, S. and J. Wyllie. 1978. Parallelism in Random Access Machines. In *Proceedings of the 10th ACM Symposium on Theory of Computing*, 114-118.

Geist, Al, Adam Benguelim, Jack Dongara, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. 1994. PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing. Cambridge, MA: The MIT Press.

Giganet Incorporated. 1999. *Giganet cLAN Family of Products*. http://www.giganet.com/products (Accessed 24 May 1999).

Grama, A., A. Gupta, and V. Kumar. 1993. Isoefficiency Function: A Scalability Metric for Parallel Algorithms and Architetcures. *IEEE Parallel and Distributed Technology* 1 (3): 12-21.

Gropp, William, Ewing Lusk, and Anthony Skjellum. 1999. *Using MPI: Portable Parallel Programming with the Message-Passing Interface, 2nd ed.* Cambridge, MA: The MIT Press.

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

Gupta, A. and V. Kumar. 1993. Performance Properties of Large Scale Parallel Systems. *Journal of Parallel and Distributed Computing* 19: 234-244.

Gustafson, J. L. 1998. Reevaluating Amdahl's Law. *Communications of the ACM* 31 (5).

Hebert, Shane L., W. Seefeld, Anthony Skjellum, Clayborne D. Taylor, and Rossen Dimitrov. 1998. MPI for NT: Two Generations of Implementations and Experience with the Message Passing Interface for Clusters and SMP Environments. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, (1): 309-317.

Intel Corporation. 2000. *Get on the fabric: IniniBand fabric demonstration.* http://download.intel.com/dsign/server /future_server_io/documents /get_on_fabric.pdf (Accessed 15 March 2001).

Li, Jin, and Anthony Skjellum. 1997. A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. *Concurrency: Practice and experience* 9(5): 345-389.

Luke, Ed, Ioana Banicescu, and Jin Li. 1998. The optimal effectiveness metric for parallel application analysis. *Information Processing Letters* - Elsevier, North Holland, Special Issue on Parallel Models 66(5): 223-229.

Kanevsky, Arkady, Anthony Skjellum, and A. Rounbehler. 1998. MPI/RT – An emerging standard for high-performance real-time systems. In *Proceedings of the 31st annual Hawaii international conference on systems sciences*, (3): 157-164.

Kumar, Vipin, Ananth Gramma, Anshul Gupta, and George Karypis. 1994. *Introduction to parallel computing: Design and analysis of algorithms*. Redwood City, CA: Benjamin/Cummings.

Kumar, Vipin and Anshul Gupta. 1990. Analyzing scalability of parallel algorithms and architectures. In *Proceedings of the 1990 international conference on parallel processing*, (2): 171-178.

Kumar, Vipin and Anshul Gupta. 1993. Performance properties of large-scale parallel systems. *Journal of Parallel and Distributed Computing* (19): 234-244.

Laudon, James and Daniel Lenoski. 1998. *The SGI Origin: A ccNUMA Highly Scalable Server*. URL:www.sgi.com (Accessed 21 May 2000).

Message Passing Interface Forum. 1998. *MPI-2 Standard*. http://www.mpi-forum.org/docs/mpi-20.ps (Accessed 24 May 2000).

Message Passing Interface Forum. 1994. *MPI: A Message-Passing Interface Standard. International Journal of Supercomputer Applications* 8 (3/4): 165-414.

Microsoft Corporation. *Supercomputing: From classics to clusters*. http://www.microsoft.com/windows2000/hpc/default.asp (Accessed 8 March 2001).

Myricom Incorporated. *Myrinet: A brief technical overview.* http://www.myri.com/myrinet/overview.html (Accessed 7 February 1997).

Myricom Incorporated. *The GM API*. http://www.myri.com/scs/GM/doc/gm_toc.html (Accessed 10 March 2001).

Motorola Corporation. 2000. *RapidIO: An embedded system component network architecture.* http://e-www.motorola.com/collateral /M951743392366collateral.pdf (Accessed 15 March 2001).

Nussbaum, D. and A. Agarwal. 1991. Scalability of Parallel Machines. *Communications of the ACM* 34 (3): 57-61.

Pakin, Scott, Vijay Karamcheti, and Andrew Chien. 1997. Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs. *IEEE Concurrency* 5 (2).

Pakin, Scott, M. Lauria, and Andrew Chien. 1995. High performance messaging on workstations: Illinois fast messages (FM) for Myrinet. In *Proceedings of the IEEE Supercomputing'95 Conference*. IEEE Computer Society Press.

Proakis, John G. and Dimitris G. Manolakis. 1996. *Digital Signal Processing: Principles, Algorithms, and Applications, 3rd ed*. Upper Saddle River, NJ: Prentice Hall.

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

Purushotham, B. V., A. Basu, P. S. Kumar, and L. M. Patnaik. 1992. Performance Estimation of LU Factorization on Message Passing Multiprocessors. Parallel Processing Letters 2 (1): 51-60.

Riesen, Rolf, Ron Brightwell, Lee Ann Fisk, Tram Hudson, Jim Otto, and Arthur B, Mccabe. 1999. Cplant. In Proceedings of the second Extreme Linux workshop at the 1999 USENIX annual technical conference.

Rothberg, Edward, Jaswinder Singh, and Anoop Gupta. 1993. Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors. In *Proceedings of the 20th International Symposium on Computer Architecture*, 14-25.

Skjellum, Anthony. 1998. High performance MPI: Extending the message-passing interface to higher performance and higher predictability. In *Proceedings of the PDPTA'98 Conference*, Las Vegas, NV.

Skjellum, Anthony, S. G. Smith, Nathan E. Doss, Alvin P. Leung, and M. Morari. 1994. The design and evolution of Zipcode. *Parallel computing, special issue on message passing* (April): 556-596.

Sohn, Andrew, Yunheung Paek, Jui-Yuan Ku, Yuetsu Kodama, and Yoshinori Yamaguchi. 1999. Communication Studies of Single-threaded and Multithreaded Distributed Memory Machines. In *Proceedings of the Fifth International Symposium on High Performance Computer Architecture*, Orlando, FL, 310-314.

Sprangers W., T. Bermmerl, S Gillich, O. Michaux, J. Hauser. 1995. Parallelization of Aerospatiale's CEL3GR Code. In *Proceedings of the Intel Supercomputer Users Group (ISUG) Conference*, Albuquerque, NM.

Sun, X. H. and D. Rover. 1994. Scalability of Parallel Algorithm-Machine Combinations. *IEEE Transactions on Parallel and Distributed Systems* 5 (6).

Tanaka, Y., M. Matsuda, K. Kubota, and M. Sato. 1998. Performance Improvement by Overlapping Computation and Communication on SMP Clusters. In

Proceedings of Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV.

Tang, Z. and G. J. Li. 1990. Optimal Granularity of Grid Iteration Problems. In *Proceedings of the 1990 International Conference on Parallel Processing*, (1): 111-118.

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

von Eicken, Thorsten and W. Vogels. 1998. Evolution of the Virtual Interface Architecture. *IEEE Computer*.

von Eicken, Thorsten, et al. 1995. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of 15th Symposium on Operating System Principles*. New York, NY: ACM Press, 40-53.

von Eicken, Thorsten, D. E. Culler, S. C. Goldstein, and K. E. Schauser. 1992. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*, Gold Coast, Australia. New York, NY: ACM Press 20 (2): 256-266.

APPENDIX A

PERSISTENT COLLECTIVE OPERATIONS INTERFACE

Interface to MPI Persistent Collective (MPIPC) Library

```
#include <stdlib.h>
#include <mpi.h>

typedef void *    MPIPC_Request;    // collective request handle type
typedef int       MPIPC_Status;     // collective status type

int MPIPC_Gather_init(
      void              *sendbuf,
      int                sendcount,
      MPI_Datatype       sendtype,
      void              *recvbuf,
      int                recvcount,
      MPI_Datatype       recvtype,
      int                root,
      MPI_Comm           comm,
      MPIPC_Request     *req        // MPI_Gather persistent request
      );

int MPIPC_Bcast_init(
      void              *buffer,
      int                count,
      MPI_Datatype       datatype,
      int                root,
      MPI_Comm           comm,
      MPIPC_Request     *req        // MPI_Bcast persistent request
      );

int MPIC_Reduce_init(
      void              *sendbuf,
      void              *recvbuf,
      int                count,
      MPI_Datatype       datatype,
      MPI_Op             op,
      int                root,
      MPI_Comm           comm,
      MPIPC_Request     *req        // MPI_Reduce persistent request
);

int MPIPC_Request_start(
      MPIPC_Request      req        // initiate persistent request
);

int MPIPC_Request_wait(              // wait on persistent request
      MPIPC_Request      req,
      MPIPC_Status      *status
);

int MPIPC_Request_free(              // release persistent request
      MPIPC_Request     *req);
```

Internal Interface for Implementing MPIPC Library

```
#include <stdlib.h>
#include "mpi.h"

#define SUCCESS         (0)
#define MPIPC_TAG       (16347)

typedef enum
{
      FALSE,
      TRUE
} bool_t;

typedef enum
{
      BCAST,
      GATHER,
      REDUCE
} coll_op_t;      // identifier for type collective operation

struct _mpipc_req_t;
typedef int (start_ft)(struct _mpipc_req_t *r); // start function type
typedef int (wait_ft)(struct _mpipc_req_t *r);  // wait function type

typedef struct _mpipc_req_t // internal representation of MPIPC_Request
{
      coll_op_t          type;
      int                np;
      int                rank;
      int                root;
      int                num_recv;
      MPI_Request       *recvreq;
      int                num_send;
      MPI_Request       *sendreq;
      start_ft          *start_fn;
      wait_ft           *wait_fn;
      bool_t             has_root;
      bool_t             is_done;
} mpipc_req_t;

typedef enum
{
      ROOT_TO_LEAVES,
      LEAVES_TO_ROOT
} tree_dir_t;     // identifier for direction of tree algorithms

typedef enum
{
      NOOP,
      SEND,
```

```
      RECV
} tree_op_t;      // type for operation in tree algorithms

typedef struct
{
      int               NumRanks;
      int               LocalRank;
      int               RootRank;
      int               NumPhases;
      int               CurrentPhase;
      tree_dir_t        Direction;
      unsigned int      Mask;
      unsigned int      BigMask;
      bool_t            IsFinished;
      bool_t            ToMyself;
} tree_t;

// prototypes of operation-specific start functions
int MPIPC_Start_gather(mpipc_req_t *r);
int MPIPC_Start_reduce(mpipc_req_t *r);
int MPIPC_Start_bcast(mpipc_req_t *r);

// prototypes of operation-specific wait functions
int MPIPC_Wait_gather(mpipc_req_t *r);
int MPIPC_Wait_reduce(mpipc_req_t *r);
int MPIPC_Wait_bcast(mpipc_req_t *r);

// interface to a binary tree object used in tree algorithms
int TreeInit(
      int         NumRanks,
      int         LocalRank,
      int         RootRank,
      tree_dir_t  Direction,
      tree_t      **Tree
);

int TreeNextOp(
      tree_t      *Tree,
      int         *PeerRank,
      tree_dir_t  *Operation
);

int TreeFree(
      tree_t      *Tree
);
```

APPENDIX B

MPI IN MULITHREADED ENVIRONMENT

Introduction

This paper discusses a parallel programming model that utilizes MPI for communication between networked nodes and multiple threads for intra-box parallelism. First, the MPI and the multithreaded models are briefly reviewed. Then, the combined programming model is introduced. Special attention is paid to the requirements and architectural features both of MPI and the low-level message-passing layer for achieving efficient parallel processing with the combined model. The distinction between thread safety and thread awareness is discussed. The document specifically concentrates on the Sandia Portals messaging interface and the MPI implementation that MPI Software Technology develops for Sandia. Because the MPI standards do not provide sufficient guidelines to describe multithreaded parallel programming, this document will also become a portability baseline for such programs.

Overview of Message Passing and Multithreading Models for Parallelism

Message passing is one of the best-understood and most widely used models for programming parallel computers. With the introduction of portable interfaces for message passing such as MPI, the cost of the parallel applications software has been significantly reduced. This has lead to further growth of the user base of parallel processing with message passing. MPI allows for porting both the parallel source code and the application's performance. MPI enables parallel programmers to design and implement algorithms at an abstract level without taking into account distinct features of the target platforms. When the parallel programs are executed on a specific platform they

can benefit from the most efficient hardware and software mechanisms available through the MPI implementation on the target platform.

In the message-passing model both the sender and the receiver participate in the message transfer, which leads to implicit synchronization between the communicating processes. This synchronization is not suitable for a number of parallel algorithms. Typically such algorithms have highly unstructured and irregular communication patterns and the synchronization imposed by message passing leads to extra overhead and performance degradation. Furthermore, algorithms that exchange a large number of short data items incur substantial overhead by message passing systems that use packet encapsulation in several levels, thus leading to low coefficient of useful data transferred.

Shared memory is the major alternative model to message passing for exploiting concurrency. Multithreading is one of the approaches for accessing the shared memory model from the programmer's viewpoint. As opposed to message passing, multithreading can be used only within a single machine. At present, multiprocessor systems are often used for cost-effective increase of the processing capabilities of high-end servers, workstations, cluster nodes, and home computers. Currently, platforms with four processors are commonly used, while some companies offer higher-degree SMP nodes, notably IBM, SGI, Sun. Further architectures with larger SMP counts are always being developed, and these become building blocks for certain of our customers.

A significant impediment for utilizing efficiently the number of processors on a single platform is imposed by the limitations of the memory subsystems. Even though the efficiency of the additional processors when executing memory intensive applications

may be relatively low, utilizing these processors is still cost effective considering the low additive price of each new CPU on the motherboard. At present, more scalable memory subsystems with higher throughput are being developed. This will improve the efficiency of the multi-processor systems. These are some of the reasons that support the hypothesis that intra-box concurrency is going to become even more important in the context of high-performance computing. An example of this trend the recently selected architecture of the 30T machine at LANL based on 32-way Alpha GS320 servers.

Using threads for intra-box concurrency[1] is probably the most natural choice, especially when the host operating system supports native threads at kernel level. Such operating systems are Sun Solaris and Microsoft Windows NT/2000. Threads offer a wide variety of schemes for constructing concurrent applications, which allows them to be used for implementing a wide range of parallel algorithms. Another strong side of threads is that their interaction with the host platform can be highly optimized considering the fact that threads are integral part of the operating system. A significant drawback of the multithreaded model for concurrency is that threads are implemented differently in different operating systems and programs that use threads are not portable. This problem has been significantly alleviated with the introduction of POSIX Threads (pthreads). Pthreads are available on most Unix operating systems. This enables concurrent programs with pthreads to be ported with minimal effort across platforms.

---

[1] Message passing with MPI provides implicit synchronization and explicit message transfer; thread programming provides explicit synchronization with implicit message transfer. These complement each other well, and threads evidently match the SMP model well. MPI always requires a single copy of a message, even on an SMP, whereas threads can share data.

Clusters of workstations interconnected with high-speed networks are rapidly becoming the platform of choice for building systems for high-performance parallel computing. Often these clusters are built with multi-processor nodes. This introduces two (or more) different media for exchanging data between any two processors on the cluster - the network interconnect and the memory bus of a multiprocessor computer node. Several approaches are available for taking advantage of the different communication media. One of the most widely used approaches is an MPI implementation that supports both a network and an SMP device. The network device provides optimal communication over the network interconnect while the SMP device utilizes possibly the most efficient mechanisms for inter-process communication offered by the host operating system. A number of available MPI implementations - among which are MPICH and MPI/Pro - have multi-device architectures. However, enabling multiple communication devices in an MPI implementation is only one of the requirements for efficient multi-device mode of operation. Another important requirement is ensuring minimal interdependency and providing high overall efficiency when all devices are engaged in communication simultaneously. This particular issue is often disregarded in the literature that discusses multi-device MPI implementations. Fundamentally, these issues are related to the internal architecture of the MPI library and the message completion and notification mechanisms.

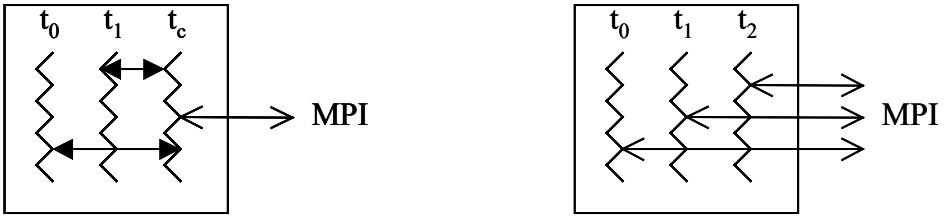
The second approach for utilizing efficiently network and intra-box communication in a cluster of SMP machines is by using threads for local communication and MPI for communicating between the nodes. This approach attempts to exploit the strong sides of both MPI and threads. Threads provide an intuitive interface

for SMP communication while MPI provides portable and high-performance communications over the network interconnect. Among the weaknesses of the combined MPI + threads solution are the increased complexity of applications' code and the potentially reduced code portability. A serious impediment for the combined approach is the fact that only few of the available MPI implementations can support multithreaded applications. Furthermore, clear guidelines of what an acceptable multithreaded MPI program is are needed. This discourages application programmers from attempting to exploit the benefits of the combined approach, since portability may be impacted, by both of these factors.

Message Passing Programs with Threads

Two main models for writing MPI programs with threads can be distinguished. These models can be denoted as *symmetric* and *asymmetric*. The asymmetric model uses one communication thread and multiple worker threads. The communication thread is dedicated to communication activities only while the worker threads perform computation. According to this model, the communication thread is the only thread that makes MPI calls. Its primary goal is to transfer data prepared by the worker threads and receive messages from remote processes. Worker threads are loosely synchronized with the communication thread through OS kernel objects (semaphores or events). Whenever a worker thread has a buffer ready to be sent, it notifies the communication thread about the availability of the message and its attributes as well as the destination process and thread. The communication thread than makes an MPI call that reflects this specification

and delivers the message to the target process with appropriate tag and logical context ID. On the receive side, the communication thread receives the message and notifies the corresponding worker thread about message arrival.



a.) Asymmetric model b) Symmetric model

Figure B.1 Models for multithreaded programming with MPI

The symmetric model does not require a specialized communication thread. Instead, all worker threads perform both communication and computation. This model presents a uniform view of all threads, which leads to a simpler code structure. Also, the symmetric model suggests higher degree of concurrency, which can be effectively utilized if the hardware and the MPI implementation can ensure concurrent progress of communication and computation. The symmetric model provides more opportunities for efficient parallel processing with multithreading than the asymmetric model. First, the extra overhead for synchronizing the worker threads with the dedicated communication thread though kernel objects is eliminated. Second, on systems with a high degree of local parallelism, the communication thread would clearly become a performance bottleneck. Third, the symmetric model can efficiently exploit the capabilities of an MPI library and networking architecture that can support concurrent communication activities.

Finally, from software engineering standpoint, the source code of the application will be easier to maintain since the threads will have symmetric implementations and functionality.

Thread Safety and Thread Awareness of MPI

Although at first glance it might be difficult to make a clear separation between thread safety and thread awareness the following definition of these terms will help in understanding important issues in the design and implementation of MPI and the low level messaging layers, such as Portals. Thread safety is defined as the quality of a software system and in particular MPI to enable programs that use multiple threads while the software system in question is in active state. In the case of MPI this would mean that the parallel application makes MPI calls between MPI_INIT and MPI_FINALIZE from multiple threads. Thread awareness is defined as a quality of the system to not only enable multithreaded programs but also provide certain optimizations such that the threads can simultaneously make independent progress while participating in communication.

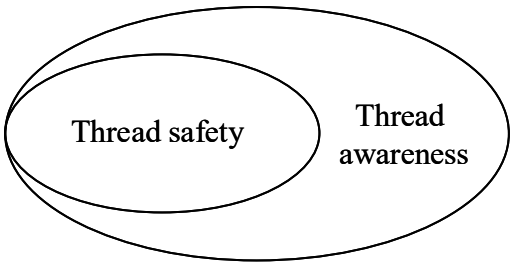


Figure B.2 Requirements for thread safety and thread awareness

Thread awareness is introduced to distinguish systems that offer complex approaches to achieving optimal multithreading mode of operation from systems that use trivial approaches such as big (or giant) locks around the entire MPI library/interface. In this context, thread awareness can be viewed as a stronger and more desirable characteristic than thread safety. Thread safety can be viewed as a sufficient requirement for supporting the asymmetric model of programming MPI with threads. In this model only the communication thread makes calls to MPI.

Consequently, even if the MPI library is thread aware and presents optimizations for multithreaded mode, these optimizations will not be utilized by the MPI application. So, for multithreaded MPI programs that use the asymmetric model, thread safe and thread aware MPI library are to large extent equivalent. On the other hand, programs written on the symmetric model will clearly benefit from the optimizations presented by the thread aware MPI implementation.

Multithreaded Semantics for Programming MPI

The MPI-1 standard does not treat the issue of multithreading. The only statement that is relevant to multithreading is that the MPI library is expected to work with multithreaded user applications. In contrast, the MPI-2 standard[2] introduces a section in the external interfaces chapter that is specifically dedicated to the interaction between MPI and threads. This section proposes an MPI_THREAD_INIT API for applications that use threads. However, the use of this API is not mandatory and is left to the

discretion of the user. The standard has consequently missed a clear opportunity to introduce a standardized interface between multithreaded programs and MPI, which could have lead to the development of MPI libraries that provide optimizations based on this interface. It is likely that because the API is optional, some early implementations of MPI-2 will not implement this API or it will be a null operation. Then, applications created for these early implementations may choose not to use the thread MPI API, thus limiting the opportunities for efficient support of multiple threads provided by more elaborate MPI-2 implementations.

Further, the MPI-2 standard leaves open the discussion on the semantics for using the MPI interface by multithreaded applications. Even when used alone, multithreading and message passing are rather complex mechanisms for concurrent processing. Correctness and determinism are always a significant concern in a concurrent system. These issues are further complicated when multithreading and message passing are used in conjunction. Consequently, an elaborate discussion on the semantics of the interaction between threads and MPI is necessary.

The only requirement that is related to the semantics of multithreaded user programs is the restriction on the multithreaded user application to use multiple "conflicting" communication MPI calls. The standard does not provide a clear interpretation of the term "conflicting". By the definition of the four levels of thread compliance, it is clear that in MPI_THREAD_MULTIPLE mode, the user program can

---

[2] The MPI-2 standard is a superset of the MPI 1.2 standard.

call the MPI library from multiple threads without any restrictions. Hence, when working in a MULTIPLE mode, the library should allow multiple concurrent communication calls too. So by inference, it can be concluded that any two communication calls should not be treated as conflicting. The semantics of what constitutes conflicting calls should be clarified before making assumptions about how MPI should support multithreaded programs. This is especially important for this work, whose goal is to go beyond thread safety and provide guidelines for an efficient cooperation of threads and MPI.

In order to facilitate its goals to study the design and internal architecture of MPI and low-level communication software such that they allow efficient use of multiple threads in applications, this work assumes the weakest interpretation of the rule for avoidance of multiple conflicting calls. According to this interpretation, any user thread can perform any MPI call regardless of what other concurrent threads might be doing at this moment. The correct semantics of these multiple MPI calls is left to the user. For example, this interpretation allows two user threads to call MPI_BCAST over the same communicator simultaneously. Evidently, the MPI library cannot guarantee that all transfers associated with one of the broadcasts will be initiated before none of the transfers associated with the other broadcast is started. If the user does not take the necessary steps, the outcome of the concurrent operations may be incorrect. Since the MPI library has no knowledge of which thread has submitted the communication requests, it cannot ensure the necessary serialization of the transfers for achieving the correct semantics of the MPI broadcast operation. The interpretation presented here allows for this concurrency in order to facilitate cases when two or mode threads can

initiate communication operations, possibly the same, over different communicators or with different user tags (for point-to-point operations). In these cases, a stricter interpretation of the conflicting rule may restrict unnecessarily the user application of exploiting concurrency that might be provided by the MPI library and low-level communication system.

Another interesting use case that demonstrates important consequences for MPI library is the case when two or more threads call MPI_WAIT on one or more requests. In one of its clarifications, the standard states the MPI library should ensure independent progress of two or more threads that wait on different requests. This automatically eliminates the possibility of using a big/giant lock around the interface and guaranteeing true blocking semantics. Since threads are allowed to complete their request regardless of the status of the other threads, each thread should acquire the lock, check for completion of the request, and release the lock in a polling loop so that the other threads can check for completion of their requests. If this is not guaranteed, deadlocks may occur. Further elaboration on this situation as well as a discussion on the more general case with MPI_ WAITANY is provided later in this document in the section that focuses on Portals.

A significantly more complex is the scenario with two or more threads waiting on the same request. In this case, in order to avoid deadlocks, the MPI library should enable all threads to continue when the request is completed. The MPI library has to keep track of how many and which threads make concurrent MPI_WAIT calls (or worse, MPI_WAITANY calls) on the same requests and then use a mechanism to signal all of them when the request is completed. If all threads return with success, this may lead to

non-determinism. A more acceptable outcome would be if only one thread returns with success and the others return from the blocking call with failure or some special status. The thread that returns with success will have its MPI_STATUS object updated accordingly. Providing a correct implementation of this semantics is a significant challenge. There are three approaches to solving the problem. The simplest of all is if the MPI library takes a restricts the conflicting rule to not guarantee that multiple threads waiting on the same request will all progress after this request is completed. Then, the user will have to make sure that the application never calls a blocking completion function on the same request. Thus, the entire responsibility for correctness is shifted to the user. The second approach requires a special device thread (may be more than one) that will make independent progress and signal the user threads. The device threads will ensure that no deadlock will occur. In the third approach, the device threads may be eliminated but the messaging layer will have to provide sufficient mechanism to guarantee that multiple user threads can make calls to synchronization objects signaled directly by the communication layer.

Requirements to MPI for Thread Safety and Thread Awareness

As a first step toward designing a thread safe and a thread aware MPI implementation a set of requirements will be defined. Often, MPI implementors proceed with development focused on usability, performance, and portability, while leaving thread safety issues for later stages of the library evolution. While theoretically feasible, this approach to MPI design leads to sub-optimal systems incorporating solutions that

primarily aim to support multithreading mode without emphasis on performance optimizations. This paper strives to identify and study important interactions between a multithreaded user program and MPI with their impact on the low-level communication software. The goal is to reach sufficient level of understanding of these interactions and consequences so a coherent set of requirements can be defined, which later can be used as a foundation of a successful design. The requirements mainly focus on protecting shared MPI or messaging layer resources as well as communication channels that might be accessed simultaneously by more than one user or MPI library internal threads.

User programs interact with MPI through opaque handles, such as MPI_COMM, MPI_DATATYPE, and MPI_REQUEST. These opaque handles have an internal representation (MPI objects) and are organized in containers. MPI provides API functions for constructing, manipulating, and releasing these objects. One clear requirement for a thread-safe MPI is providing atomic creation and destruction of the internal objects represented by the opaque handles. For example, MPI has to allow multiple user threads to create derived data types by calling simultaneously MPI_TYPE_STRUCT. In a similar fashion, user threads should be allowed to construct communicators, key values, and requests. It is the MPI implementation's responsibility to provide sufficient protection of the internal objects and containers to enable thread safety in this aspect[3]. The constructors

---

[3] It is desirable to use optimistic locking of objects, rather than pessimistic (overkill) locking. Information about whether a program is multithreaded, or whether specific instances of MPI objects are used in multiple user threads are needed for optimistic schemes. The semantics of MPI, including the optional nature of thread-related calls, weakens the opportunity to do locking optimistically, without using out-of-band profiling or other types of restriction of the semantics of the user program, either through dynamic discovery of its emergent semantics, annotation, or by repetitive observation generating feedback to the library for subsequent runs.

and destructors of all objects with the exception of the request object are invoked relatively infrequently and they do not require optimizations for multithreading.

A more challenging task is the creation and manipulation of MPI_Request objects. Throughout the lifecycle of a parallel application, whether single threaded or multithreaded, a large number of requests are generated, stared, waited or tested on, and released. The START and WAIT/TEST operations are especially sensitive to overheads so they should be carefully designed for thread safety. At the same time, these operations provide the largest opportunity for thread awareness optimizations. A trivial approach to these sensitive operations would be to serialize them with locks, but then programs written in the symmetric model will not actually exhibit any significant benefit of the potentials for concurrent communication.

In a multithreaded MPI program that uses the symmetric model, more than one thread per MPI process can participate in communication, which explicitly or implicitly can lead to concurrent transfers to or from the same peer process. Since the library has no knowledge of the number of user threads, these threads cannot be assigned specific tags that might be used for creating safe communication spaces in MPI. A safe communication space in MPI is defined only over a communicator. User tags can be used to present a two-dimensional view of this space, but a program must intentionally adopt such a convention.

The active component in MPI is the process and no allowances for thread identifiers are made. Consequently, two user threads that belong to the same process may initiate a send operation to the same destination. In such situations a thread-safe MPI

library should ensure atomicity of message transfers so that bytes of the two concurrent messages to the same destination do not get mixed. The atomicity of the transfers can be achieved using several approaches. The distinction of these approaches is based on the semantics and capabilities of the lower communication layers that are used by MPI for transferring the messages as well as the thread safety/awareness of these layers.

The classification of the messaging layers is made from MPI's point of view. The first group of messaging systems is passive. Such communication layers are TCP, VIA, and Myrinet GM. TCP is a stream oriented protocol and is implemented in the operating system (kernel calls are used through the OS API) while VIA and GM are message oriented and rely on OS bypass mechanisms with intelligent network controllers; most VIA and GM calls do not generate kernel context switches. Regardless of the implementation, all of the above mentioned systems are considered passive for the purposes of this discussion. Passive layers can be viewed primarily as data movers. When used with passive layers, MPI participates actively in all phases of the protocols as well as matching and progress. Since multiple threads may access the same socket or VI connection at the same time, MPI has to lock these channels in order to ensure atomicity of the message transfers[4]. Myrinet GM is not thread safe by definition, so a special care must be taken for all calls to the GM API. MPI implementations with internal threaded architectures, such MPI/Pro for VIA and TCP will take some of the user thread-safety measures even in a single threaded user processes. These implementations use a system

---

[4] MPI provides pair-wise ordering between pairs of processes, per communicator.

thread that ensures independent progress of messages and asynchronous completion of requests. Such implementations can support multi-device mode of operation as well as multiple simultaneous MPI_WAIT calls from different threads. This is achieved by delegating the synchronization interface between the messaging layer and MPI to the system thread. Then, user threads can block on an event associated with each particular request, which allows independent completion of the requests. MPI layered on top of a passive messaging layer is obligated as a minimum to protect the communication channels built by the communication layer and possibly the entire API. In both cases the requirements to the passive layer for achieving a thread-safe MPI are minimal. Thread-safety is implemented by MPI itself.

The active messaging layers, such as Sandia Portals and Los Alamos ULM, raise the abstraction of the interface between MPI and the messaging layer by not only performing data movement but also progress, matching of incoming messages, buffer management, and protocol implementation. These messaging layers facilitate a low-overhead and scalable MPI and at the same time provide sufficient set of primitives that suggest an MPI implementation without an internal system thread. In the context of thread safety, this means that the MPI implementation may delegate some of the thread safety concerns to the messaging layer, thus increasing the potentials for internal concurrency and efficient processing in multithreaded mode. As a consequence, MPI does not need to perform explicit locks over the channels and synchronizations objects/primitives. This will enable multiple user threads to initiate communication activities or synchronization for request completion concurrently, which as defined

earlier is considered a feature of thread awareness. Consequently, for achieving an efficient multithreaded mode of operation of MPI applications, both the MPI library and the underlying communication layer has to cooperate in providing maximum concurrency in the accesses to latency sensitive shared data structures, communication channels, synchronization objects, and system buffers.

The final thread safety requirement discusses the impact of the mechanisms for ensuring thread safety and thread awareness on programs that do not use multithreading. This is especially important for MPI implementations that do not have thread architecture, so they do not need to have internal protection of the shared data structures and communication channels in a single threaded user application mode. Such MPI implementations are MPI/Pro for Portals and MPI/Pro for ULM. These MPI libraries should allow single-threaded MPI programs to operate without incurring overhead necessary for ensuring thread safety. Since almost all of the legacy MPI codes are single-threaded, thread-enabled MPI libraries should provide adequate support for these applications without performance degradation caused by multithread-oriented optimizations that might be present in these libraries for the benefit of other applications. For example, the MPI implementation developed by MPI Software Technology for Sandia Portals uses a mechanism that allows the library to work in a mode that bypasses all of the locks and kernel calls for synchronization. This mode can be selected in run-time through an environment variable set by the user or by a call to the MPI-2 API function MPI_THREAD_INIT(). By setting the environment variable or calling the function the user waives the requirement for thread safety and promises not to use

multiple threads in his application. Thus, single-threaded application will not incur extra overhead introduced in the library by the multithread mechanisms[5].

Analysis of a Thread-Aware MPI Design for Portals

Assumptions

This discussion assumes the use of the Portals 3.0 API in the implementation of a library conforming to MPI 1.2 and supporting multi-threaded MPI applications. In particular we discuss design decisions based on the "Thread-aware MPI for Sandia Cplant" implementation by MPI Software Technology, Inc.

The MPI implementation should not require any internal threads of its own to cleanly support multi-threaded applications over Portals. The MPI library need only support a single Portals interface (NI).

NOTE: This section assumes a high degree of familiarity with the Portals API and in particular the MPI section of the paper “The Portals 3.0 Message Passing Interface”. Readers can find this document at http://www.cs.sandia.gov/~bright.

Goals

The MPI library should strive for a high degree of internal concurrency in order to allow the user's threads to execute MPI calls as independently as possible without blocking on one another unnecessarily, which may hurt efficiency. Allowing requests to complete as independently as possible also helps avoid deadlock situations that may

---

[5] This type of solution is acceptable provided the user application knows that no libraries use threads internally. In

occur if multiple requests' progresses are highly interdependent in complex (possibly non-local) ways.

The implementation of all MPI calls that are defined as "blocking" should block "nicely", that is, use a fixed amount of overhead to complete regardless of the amount of wallclock time spent in the call. This allows the programmer to use the maximum CPU time available to his multi-threaded application.

Request Initialization

Initializing a send or receive request currently involves little interaction with Portals. Portals doesn't require any heavy-weight operations like pinning of the user memory that can be performed at request initialization.

NOTE: There is an opportunity to bind the memory descriptor for a short-protocol send to the NI at initialization time with PtlMDBind(). This is currently being done at start time.

Request Activation

Send

Depending on the protocol being used for the send, the sender either posts a match entry (ME) to the global Portals match list and attaches the memory descriptor (MD) to it (rendezvous protocol) or just binds the MD directly to the NI (eager protocol). The sender then starts the transfer by calling PtlPut(). There is only one match list

---

other words, the single-threaded behavior of the application is a global property, not a property per communicator.

associated with all send operations of an MPI task, regardless of communicator or destination rank.

There is no need for the MPI library to actively "make progress" on sends. The data transfer can be completely handled by Portals and the receiver once the send has been posted: the receiver pulls data directly from the sender via PtlGet() during the data transfer phase of the rendezvous protocol. There is no need for a sender thread to participate, so the event queue only needs to be checked when testing for the start (PTL_EVENT_SENT) or completion of a long send request (PTL_EVENT_GET or PTL_EVENT_ACK).

NOTE: separated send and recv event queues would mean a thread could not MPI_WAITANY on a send and recv request without polling both queues. More about this below in the “Completion” discussion.

Expected Receive

A data transfer is "expected" from the receiver's point of view if the matching receive request has already been posted when the sender starts the send. If the send hasn't come in yet, the receiver inserts a ME into the recv match list and attaches the appropriate MD to it with the MD's threshold = 0. The receiver than uses PtlMDUpdate() to atomically enable the ME while making sure the matching send doesn't come in at the same time. When the matching send arrives Portals automatically returns an ACK back to the sender if needed and disables the receiver's ME. The receiver eventually finds a PTL_EVENT_PUT in his event queue and knows that the data has arrived.

Unexpected Receive

A data transfer is "unexpected" for the receiver if the send starts before the receive request has been posted. If the eager protocol is being used for a short message, the incoming data will match the ME for a preposted unexpected receive buffer. As far as Portals is concerned this is very similar to the expected case discussed above. Eager protocol for a long message is detected as an error by MPI.

When the long protocol is used, the incoming data will skip the preposted buffers and leave only a PTL_EVENT_PUT in the event queue. Eventually the receiver will find the event and do a PtlGet() to fetch the data from the sender's ME, followed by a PTL_EVENT_REPLY that indicates all the data has been received.

Request Completion

Request completion is the most interesting part of the request lifecycle in a thread-aware MPI implementation because the calling thread may block in a completion call until one or more requests are complete. Allowing a thread to block efficiently so other threads may run while at the same time maintaining lowest possible latency for all threads is a significant challenge. The semantics of the underlying network API directly influence how this problem can be approached.

NOTE: At time of writing the thread-safety semantics of Portals are still somewhat undefined. The current MSTI implementation of MPI for Portals serializes all calls to the Portals API by maintaining a single lock (the “big Portals lock” or “BPL”)

that any thread must acquire before calling Portals. Discussion below describes the limitations of the BPL design.

Test

A non-blocking test for completion like MPI_TEST is the most straightforward of the completion calls. A thread calling MPI_TEST tries to acquire the BPL and if successful, polls the NI for incoming events with PtlEQGet(). Events are passed to the protocol-handling parts of MPI, which may eventually mark a request complete. The thread then releases the BPL, checks for completion of the request in question and returns.

Wait

MPI_WAIT is the simplest of the blocking completion calls, since it blocks the calling thread based only on the status of a single request. In single-threaded mode it is a straightforward matter to implement MPI_WAIT by having the MPI application block on calls to PtlEQWait() until the event-handlers mark the interesting request complete. In a multi-threaded application this is not safe to do, since the thread calling PtlEQWait() holds the BPL, preventing any other threads from using Portals. This leads to obvious deadlock situations.

The current MPI implementation therefore uses PtlEQGet() instead of PtlEQWait(), which means it has to constantly acquire the BPL, poll Portals, release the BPL, and yield to the scheduler until the request is complete. This behavior is not thread-

aware, since it wastes CPU cycles polling for completion instead of blocking cleanly until something happens in the network.

If we can assume that Portals allows ME manipulation concurrently with event queue access, we could break the BPL apart and create a separate lock for the event queues. This would allow some threads to nicely block in MPI_WAIT while others post send and receive requests. Unfortunately this creates an ordering dependency among requests, since they may block trying to get the event queue lock while another thread waits for an unrelated operation to complete. This can lead to subtle deadlock situations that may be hard to debug.

Assuming again the above, if MPI used a separate event queue for every request instead of a single global queue for all operations it may be possible to cleanly wait on a single request, since a request may only be waited on by one thread at a time. All threads would then only have to share an event queue for unexpected messages.

Waitany

MPI_WAITANY is the most demanding of the blocking completion calls. All of the other calls in the MPI_WAIT family can trivially be implemented in a thread-aware fashion on top of a thread-aware MPI_WAITANY. The difficulty with this call is that it is desirable for the thread to block waiting for completion of any one of several requests. These requests may involve communication to completely separate endpoints over distinct communicators.

MPI_WAITANY is currently implemented with the same naive polling loop described above for MPI_WAIT. If the implementation were to adopt the event-queue-per-request strategy described for MPI_WAIT, MPI_WAITANY would still have to be implemented as a polling loop because PtlEQWait() gives the ability to block on a single event queue only. The best solution for MPI_WAITANY with the current Portals API is to have a single unified event queue that anyone can wait on, but we are still faced with the locking problems described for MPI_WAIT.

Table B.1 Implementation of MPI communication calls with Portals

MPI Call	**Portals Calls Used**
MPI_SEND_INIT	PtlMDBind
MPI_RECV_INIT	none
MPI_START (send)	PtlMEInsert, PtlMDAttach, PtlPut
MPI_START (recv)	PtlMDBind, PtlGet, PtlMEInsert, PtlMDAttach, PtlEQGet, PtlMDUpdate
MPI_TEST	PtlEQGet
MPI_WAIT	PtlEQWait (currently PtlEQGet)
MPI_WAITANY	PtlEQWait? (currently PtlEQGet)

Requirements to the Low–Level Messaging Layer (Portals) for Thread Safety and Thread Awareness

As can be seem from the design overview above, the current MPI implementation over Portals is non-optimal for multithreaded MPI applications because of blocking

completion operations that are not thread aware. It does not seem to be much of a burden to serialize calls to set up memory descriptors or match entries, or even calls to PtlPut() or PtlGet(). However the event queues need to have well-documented behaviors in the face of multi-threaded applications.

The shortcomings of the current MPI_WAIT may be solvable by setting up a receive queue for every posted request. For this to work, Portals would need to support an arbitrarily large number of event queues and also allow multiple threads to concurrently call PtlEQWait() on distinct event queues. For MPI_WAITANY to work efficiently, Portals should provide a select()-style call that allows a single thread to wait on multiple event queues for the next incoming event.

One possibility that has been considered is that it may be useful to allow multiple threads to concurrently wait on the same event queue. This would allow threads calling MPI_WAITANY to wait on the event queue without taking a lock, avoiding the deadlock problem discussed above. It would be sufficient for the purposes of MPI to have semantics as follows for multiple threads in PtlEQWait(): *If multiple threads are blocking in PtlEQWait() and an event arrives, one of the threads will receive the event and the others will receive PTL_EQ_EMPTY, just as if a call to PtlEQGet() had been made on an empty queue.* Of course, for single-threaded applications the calling thread would always get the event, so the extended PtlEQWait() semantics would boil down to be the same as documented.

The problem with multiple threads waiting on the same queue is outlined in Figure B.3. Thread A has no way of knowing that Thread B received the event that it was

waiting for, so it goes back into PtlEQWait and doesn't check again until another event comes. It should be noted that MPI_WAITANY is much less commonly used than MPI_WAIT. A design that results in a thread-aware MPI_WAIT but a non-optimal MPI_WAITANY may be considered acceptable by most users.

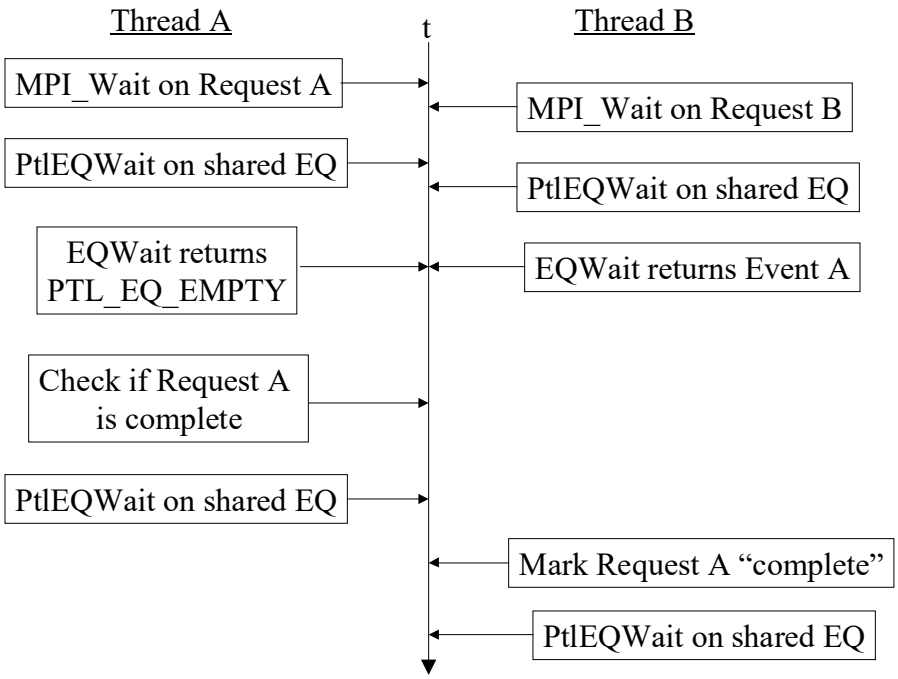


Figure B.3 Multiple threads waiting on one queue

There are other opportunities for parallelism within the MPI implementation. If it were possible to share unexpected buffers between match lists, MPI could use a different portal index for each communicator so that multiple threads could make independent progress over communicators that are as independent as possible.

Enabling Efficient Multithreading in MPI Applications

This section aims to provide guidelines for writing efficient MPI programs that use multiple threads.

Task parallel programs may share a communicator with different threads

If multiple send and/or receive threads per process exist, the use of tags is a good way to pass messages between such multithreaded processes. Because efficient MPI implementations may optimize for a small number of message tags, the number of such tags should be kept small. Without loss of portability, it may also be more efficient to pick low numbered tags, as these may be those that are the optimized small number of tags in some implementations. Blocking send and receive are best used in the multiple threads, rather than launching non-blocking calls in the multiple threads. If persistent communication is used, this will unfortunately be non-blocking, but may still be better because of performance optimizations possible in good MPI's when early-binding send/receive are exploited.

Data parallel programs should use one communicator per collective operation, and multiple communicators

In parallel to the use of multiple threads with a single communicator for point-to-point operations, one collective operation may be posed at a time in a communicator. The user program must serialize the use of these (a reasonable optimization by MPI implementations will prevent a sequence of collectives from getting confused). Using

MPI_Comm_dup, additional equivalent communicators can be created before they are needed, in order to allow multiple collective operations to be posed by multiple threads across a set of MPI processes.

MPI Programs should use available thread assertions of MPI-2

Threads should use MPI_Thread_init() when that API addition is supported, in order to help MPI distinguish multi-threaded and single-threaded applications.

MPI programs should avoid WAIT/TEST on same requests

Because the MPI standards are not clear on the allowability of waiting on the same request, it is suggested to avoid such multithreaded programs. If an implementation supports multiple threads waiting on the same request, so that one gets an affirmative result, and the others gets an error, then this capability will almost certainly come at the cost of performance.

A design lemma of the MPI Forum has traditionally been that serialization of multiple threads per MPI process should result in a correct MPI program, without new behavior. Under such an interpretation (strict), a program that in multiple threads waits on the same request would be termed "erroneous" and not "unsafe" according the MPI nomenclature.

APPENDIX C

SPECIFICATION OF EXPERIMENTAL CONFIGURATIONS

1. SAG cluster (annotated with label *sag* in the dissertation):

Cluster nodes:
CPU: 1 x Intel Pentium II @ 350 MHz
RAM: 288 MB
Chipset: 440 BX
Operating system: dual-boot Windows NT 4.0 and RedHat Linux 6.2

Network-1: Giganet cLAN:
switch: 8-port GNX 5000
NIC: 8 x GNN 1000, rev. C
link-rate: 1.25 Gb/sec

Network-2: Ethernet:
switch: 12-port 3Com PowerStack
NIC: 3Com 59x
link-rate: 100 Mb/sec

2. DIM cluster (annotated with label *dim* in the dissertation):

Cluster nodes 8:
Type: Dell Precision 410
CPU: 1 x Intel Pentium III @ 733 MHz
RAM: 128 MB
Chipset: 440 BX
Operating system: dual-boot Windows 2000 and RedHat Linux 6.2

Network-1: Giganet cLAN:
switch: 30-port GNX 5030
NIC: 8 x GNN 1000, rev. C and rev. D
link-rate: 1.25 Gb/sec

Network-2: Ethernet:
switch: 30-port 3Com PowerStack
NIC: 3Com 59x
link-rate: 100 Mb/sec

3. AC3 cluster (annotated with label *ac3* in the dissertation):

Cluster nodes 64:
Type: Dell PowerEdge 6250
CPU: 4 x Intel Pentium II Xeon @ 450 MHz

RAM: 4096 MB
Chipset: 440 BX
Operating system: Windows 2000

Network: Giganet cLAN
switch: a 64-node fabric built with 32-port GNX 5030
NIC: 64 x GNN 1000, rev. C and rev. D
link-rate: 1.25 Gb/sec

APPENDIX D

GLOSSARY OF NEW CONCEPTS AND TERMS

Methodology for classification of message-passing libraries according to their methods for message completion notification and message progress

This methodology identifies two methods for message completion notification: asynchronous (blocking) or synchronous (polling). The message progress is also classified as asynchronous (independent) or synchronous (polling). According to the classification methodology, four categories of message-passing libraries are consequently identified: {polling notification, polling progress}, {polling notification, independent progress}, {blocking notification, polling progress}, and {blocking notification, independent progress}. The commonly used MPICH implementation of MPI is an instance of an all-polling library. MPI/Pro provides options to users to select the behavior of MPI/Pro. Specifically, MPI/Pro supports a mode with blocking notification and independent progress, an all-polling mode for short messages (similarly to MPICH), and a mode with polling notification and independent progress for long messages.

Model for parallel computation based on performance attributes as observed by user processes – Bandwidth and Overhead [based parallel processing] User-level Model (BOUM)

A parallel programming model that provides for explicit description of early binding and overlapping of communication and computation. The model has been used in this dissertation to predict the performance improvement of parallel algorithms that use early binding and overlapping. The accuracy of the model is

verified through an experimental procedure, presented in Chapter V of the dissertation.

Degree of persistence

A performance metric that captures the capability of a communication system to support effective early binding. Using this metric, parallel software designers can predict the actual benefit of applying early binding to algorithms. Also, system software designers can use the metric to evaluate the quality of their implementation on a particular hardware platform and operating system.

Degree of overlapping

A performance metric whose goal is to facilitate quantitative analysis of the capacity of a parallel system to deliver maximum effect of overlapping of communication and computation to application processes. This metric is an important tool for estimating the performance benefits of overlapping on a particular parallel system. This metric captures a number of system features that are difficult to analyze but at the same time significantly affect the efficiency of overlapping. These range from purely hardware features of the system such as memory bandwidth and peripheral-bus throughput to purely software features, such as the architecture of the message-passing middleware.

Segmentation efficiency

This metric has been introduced to assist the designers of parallel algorithms that employ overlapping of communication and computation. A common approach for implementing overlapping is by breaking large messages into smaller segments that can be pipelined and overlapped with computation. The effectiveness of the segmentation procedure depends on the ratio between the overhead and bandwidth components of the communication time for the segments on the target platform. Segmentation efficiency provides a guideline for determining the optimal number of segments that yields maximum overlapping.

Degree of asynchrony

A performance metric has also been introduced to present a quantification analysis of the capability of a system to move user data while user processes are performing activities unrelated to communication. This capability is an important prerequisite for effective overlapping. This metric can be used in order to assess the capabilities of different architectures of the message-passing middleware to support asynchronous message progress. In this dissertation, the analysis based on degree of asynchrony showed that libraries that use polling progress exhibit sub-optimal behavior when user processes attempt to schedule concurrent communication and computation activities.