5-1-2010

A protocol reconfiguration and optimization system for MPI

Manjunath Gorentla Venkata

Manjunath Gorentla Venkata

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

[illegible], Chairperson

[illegible]

[illegible]

[illegible]

A Protocol Reconfiguration and Optimization System for MPI

by

Manjunath Gorentla Venkata

M.S., Computer Science,
University of New Mexico, 2005

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

May, 2010

Dedication

To my parents for many sacrifices they made for me, and for instilling the values of hard work and learning.

Acknowledgments

This research and dissertation would not have been possible without the support and mentorship of my advisor Professor Patrick Bridges. I would like to thank him for his invaluable support as a teacher, mentor, and research advisor. I would like to thank Professor Barney Maccabe for his insights and mentorship during my graduate school. I would also like to acknowledge other members of my dissertation committee, Dr. Rolf Riesen, Professor Dorian Arnold, and Professor Nasir Ghani for their feedback and support.

I would like to thank my parents, brother, and sister for their relentless support for all my endeavors. I thank my wife Shruti for encouraging me to be my best during the last years of graduate school. I would like to acknowledge many friends for their help, encouragement, and support during my years as a graduate student.

I would like to thank all past and present members of Scalable Systems Labs at the University of New Mexico for their support and friendship. Particularly, I would like to thank James Horey, Kurt Ferriera, and Edgar Leon for many interesting and lively conversations that helped me improve my research skills and ideas. I would like to thank Patrick Widener for his advise that helped me to improve my technical writing and presentation skills.

Finally, I would like to thank the Sandia University Research Program (SURP) and Department of Energy for their financial support.

A Protocol Reconfiguration and Optimization System for MPI

by

Manjunath Gorentla Venkata

ABSTRACT OF DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

May, 2010

A Protocol Reconfiguration and Optimization System for MPI

by

Manjunath Gorentla Venkata

M.S., Computer Science,
University of New Mexico, 2005

Ph.D, Computer Science, University of New Mexico, 2010

Abstract

Modern high performance computing (HPC) applications, for example adaptive mesh refinement and multi-physics codes, have dynamic communication characteristics which result in poor performance on current Message Passing Interface (MPI) implementations. The degraded application performance can be attributed to a mismatch between changing application requirements and static communication library functionality. To improve the performance of these applications, MPI libraries should change their protocol functionality in response to changing application requirements, and tailor their functionality to take advantage of hardware capabilities.

This dissertation describes Protocol Reconfiguration and Optimization system for MPI (PRO-MPI), a framework for constructing profile-driven reconfigurable MPI libraries; these libraries use past application characteristics (profiles) to dynamically change their functionality to match the changing application requirements. The framework addresses the challenges of designing and implementing the reconfigurable MPI libraries, which include collecting and reasoning about application characteristics to drive the protocol reconfiguration and defining abstractions required for implementing these reconfigurations. Two prototype reconfigurable MPI implementations based on the framework – Open PRO-MPI and Cactus PRO-MPI – are also presented to demonstrate the utility of the framework.

To demonstrate the effectiveness of reconfigurable MPI libraries, this dissertation presents experimental results to show the impact of using these libraries on the application performance. The results show that PRO-MPI improves the performance of important HPC applications and benchmarks. They also show that HyperCLaw performance improves by approximately 22% when exact profiles are available, and HyperCLaw performance improves by approximately 18% when only approximate profiles are available.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern HPC applications, such as adaptive mesh refinement and multi-physics codes, have diverse and dynamic communication characteristics. Their message characteristics (size, type, frequency) and communication characteristics (number of processes participating, type of collective operation, communication to computation ratio) change during runtime. As a consequence, these applications' communication demands constantly change.

Profile-driven reconfigurable communication libraries are one way to deal with changing application communication characteristics. They improve HPC application performance by changing their protocol functionality to match changing application communication demands. They use past application characteristics to drive these changes. For example, when the number of communicating processes changes during application execution, these libraries improve application performance by choosing transport protocols and other functionality based on the number of communicating processes.

Implementing reconfigurable communication libraries poses many challenges. To reconfigure, the libraries need to collect and reason about changing application communication requirements and environment information (runtime state, resource availability and hardware characteristics). The reconfiguration should not disrupt application execution or correctness of the solution. Further, reconfigurable libraries should be designed to support function reusability and extension.

This dissertation presents Protocol Reconfiguration and Optimization system for MPI (PRO-MPI), a framework for constructing reconfigurable libraries for MPI-based applications. The framework collects and analyzes the application communication characteristics required for a reconfiguration. It provides mechanisms to execute reconfigurations. It provides abstractions for implementing the communication libraries that supports reusability and extension of library functionality, and addition of new reconfigurations. In addition to presenting the design of the framework, the dissertation presents two MPI implementations based on PRO-MPI, and evaluates the implementations.

1.1 MPI Overview

MPI is a specification for message passing libraries used for communication by parallel applications [1, 2, 3]. It is designed to achieve low-latency and high-bandwidth communication on distributed memory machines including supercomputers, clusters, and networks of workstations. It specifies an API and its mapping to different programming languages such as C, C++, and Fortran. There are many MPI library implementations: IBM's MPI for Blue Gene/L, Cray's MPI for XT4, Myricom's MPICH2-MX , Open MPI, MPICH, MVAPICH2, Microsoft MPI, SiCortex MPI, HP MPI, and NEC MPI.

MPI provides various communication operations for communication among the processes of a parallel application. In this section we describe point-to-point, collective, and one-sided operations that are relevant to this research. More details on these operations and other MPI operations can be found in [3] and [1].

1.1.1 MPI Point-to-point Operations

MPI point-to-point operations are a set of routines to perform one-to-one communication between any two MPI processes. One process sends data or control information using the MPI_Send operation, and the other process receives this information using the MPI_Recv operation. Both MPI_Send and MPI_Recv operations use *source* and *destination* parameters to identify the *source* and the *destination* MPI processes, and the *tag* parameter for demultiplexing the MPI message.

MPI specification provides variants of MPI_Send and MPI_Recv operations covering a wide-range of semantics which include, besides the standard send, a buffered send, a ready mode send, and a synchronous send. The standard send operation allows a sender to post a send operation irrespective of the receiver status (a matching receive on the receiver is posted or not), and this operation is completed once the message is safely stored. After the send operation is completed, the sender is free to reuse the send buffer. Like the standard send, the buffered send operation also allows the sender to post a send operation and complete the operation irrespective of receive status. However, unlike the standard send, the buffered send is a local operation. Unlike the standard and buffered send, the ready mode send operation allows a sender to post a send only after the corresponding receiver is ready (a matching receive on the receiver should be posted). In synchronous send semantics, a send operation is completed only after a matching receive is posted and the message is received by the receiver. Besides these blocking operations, the MPI specification includes variants of send and receive which are non-blocking. The non-blocking send and receive operations enable the applications to overlap computation and communication.

MPI implementations provide various protocols for implementing the point-to-point operations [4, 5, 6]. Particularly, they provide an eager protocol which is optimized for small message transfer, and rendezvous and eager rendezvous protocols which are optimized for large message transfer. While using the eager protocol, the sender starts sending a message to the receiver whether or not a matching receive is posted. If a matching receive is posted, the message on arrival is copied to an application buffer, otherwise the message is buffered at receiver until a matching receive is posted. While using the rendezvous protocol for message transfer, before sending any message, the sender sends a Request to Send (RTS) message along with the message header to the receiver. If a matching receive for the message is already posted (preposted receive) [1], the receiver sends a Clear to Send message (CTS) to the sender. If a matching receive is not posted, the receiver waits for a matching receive to be posted before sending a CTS message. The sender, on receiving the CTS message, sends the data of the message. While using the eager rendezvous protocol, the sender expecting a matching receive already posted by the receiver sends both header and data of the message. If a matching receive is not posted, the data of the message is discarded, and only the header of the message is stored at the receiver. After a matching receive is posted by the receiver, it sends a CTS message to the sender, and the sender responds with resending the message data.

1.1.2 MPI Collective Operations

MPI collective operations are a set of routines to perform one-to-many, many-to-one, and many-to-many communications among a group of MPI processes. All MPI collective operations are executed on one or more communicators. A communicator, in the context of collective operations, is a group of MPI processes. When a collective

[1]Preposted receive - a receive posted by a MPI process before the arrival of the corresponding message.

operation is executed on the communicator, all processes in this communicator will participate in the collective operation.

The main collective routines in the specification are MPI_Barrier, MPI_Broadcast, MPI_Scatter (Gather, Reduce), and MP_AllGather (Reduce_Scatter, Alltoall). The MPI_Barrier routine allows a group of processes to synchronize. For many-to-one and one-to-many communications, MPI processes use MPI_Broadcast (Scatter, Gather, Reduce) routines, and for all-to-all communications, MPI processes use MPI_AllGather (Alltoall) routines.

1.1.3 MPI One-sided Operations

MPI one-sided operations are a set of routines to perform operations on a remote memory (window). In the one-sided operations, an origin process initiates a operation on a window, and a target process hosts the window. The origin process provides all information (process address, memory address) required for the operation. As a result, the target process need not be involved in the operation.

MPI one-sided specification provides MPI_Get, MPI_Put and MPI_Accumulate routines for data transfer between any two MPI processes. However, unlike the point-to-point operations, these operations do not require the receiver process to be involved during the data transfer. MPI_Get is used to read data from a window (remote memory), MPI_Put writes data into a window, and MPI_Accumulate updates a window with the local memory buffers.

1.2 HPC Environment

In recent years, the use of HPC systems has become more common. Traditionally, HPC was used to study scientific problems such as isolated black hole evolution, supernovae simulation, protein sequencing, and weather forecasting. Recently, however, HPC is being used in the retail industry for supply chain management, in car manufacturing to perform crash test analysis, finance for options equity trading, and in bioinformatics for protein folding [7, 8].

With wider application of HPC and affordability of HPC hardware, HPC machines are now more varied in their architecture. For example, more HPC machines are built from off-the-shelf components because of the rising performance and plunging costs of commodity processors and I/O hardware. The variation in architecture becomes even more evident when examining the Top 500 list [9]. The machines vary in their capabilities, services provided, and resources required for their operation.

As a result of the wider application of HPC and variation in hardware architecture, HPC application characteristics, specifically communication characteristics, are moving away from traditional uniform and static patterns to more diverse and dynamic patterns. The remainder of this section elaborates on application communication characteristics and HPC hardware characteristics.

1.2.1 Dynamic HPC Application Communication Characteristics

HPC applications generally use parallel processing techniques to solve a problem. The processes communicate with each other, exchanging data and intermediate solutions, to reach a solution for the problem. HPC applications typically use explicit communication models such as MPI for this communication.

Many traditional HPC applications using MPI libraries to have static communication characteristics: messages size, type and frequency remain similar during different phases of application, fixed set of processes communicated during application phases, and computation and communication ratio is predictable, and resource usage is predictable [10].

With wider use of HPC, however, many modern HPC applications have dynamic communication characteristics. These dynamic characteristics are the artifacts of physical system modeled by the application, the mathematical model used, or the algorithms used in the applications. Adaptive Mesh Refinement (AMR) applications, multi-physics codes, and applications using linear solvers are all known to exhibit these characteristics [10, 11].

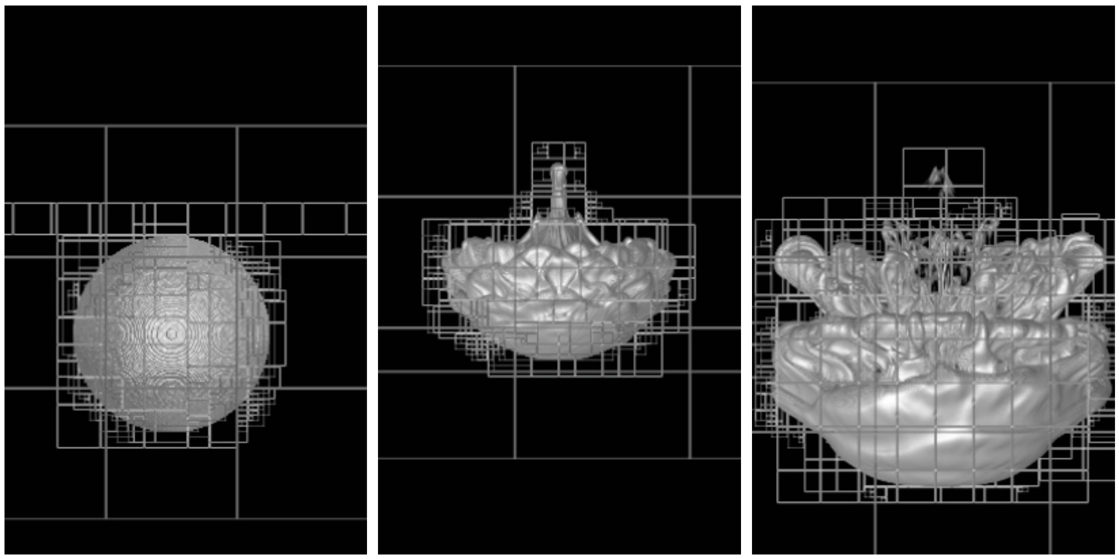


Figure 1.1: HyperCLaw's communication characteristics: Grids of irregular shape and different size

HyperCLaw is an example of an AMR application that exhibits dynamic communication characteristics. Like other AMR applications, HyperCLaw uses AMR for modeling physical systems. Modeled physical systems are represented by partial differential equations (PDE) on a computational grid. To reach the solution and conserve resources, they refine only parts of the grid that are of scientific interest, resulting in grids of irregular shape, size and resolution. Typically, each grid point is associated with a process which is responsible for the grid point computation. These processes communicate to exchange data during grid refinement and to exchange intermediate solutions. These irregular and dynamic communications result in changing communication characteristics.

Figure 1.1 shows HyperCLaw simulating the deformation of a helium bubble when it is hit by a shock wave. During simulation, the portion of the grid where the bubble is represented is refined, while the rest of the grid is not refined. Figure 1.2 and 1.3 show the communication characteristics of HyperCLaw. These figures show that that both the processes participating in the communication and process pairs exchanging messages vary during the application execution.

1.2.2 HPC Hardware Chracteristics

HPC applications run on machines that vary from clusters built from off-the-shelf components to supercomputers built from proprietary components. These machines vary in architecture, hardware capabilities, services provided, and resources used. For example Red Storm, a Cray XT4 machine at Sandia National Laboratories, is used for weather modeling, ocean modeling, and other classified research; it provides demultiplexing of messages to high-level protocols on the network interface, guarantees memory for every allocation, and uses a low-noise operating system to provide latency guarantees. In contrast, Encanto, a machine in the New Mexico Computing Applications Center (NMCAC), is used for energy and environment research; it is built from commodity components such as SGI Altix nodes and Infiniband network

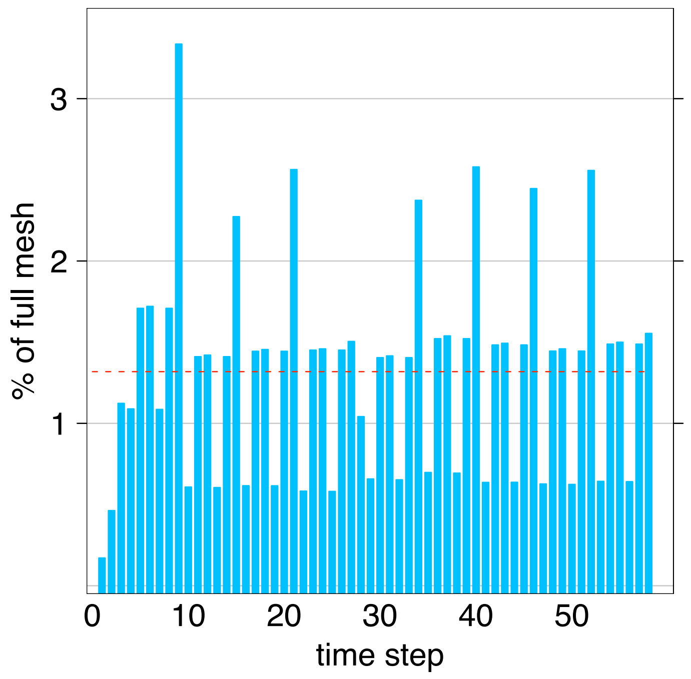


Figure 1.2: Communication characteristics of HyperCLaw showing the percentage of total processes participating in a communication during each application time step.

interfaces, and does not provide many of the capabilities that Red Storm provides.

1.3 Demands on Communication Libraries

Communication libraries provide communication services to satisfy applications communication requirements. They abstract the services provided by the network hardware, and add functionality that is required by applications but not provided by the network hardware. As a result of its role, communication library functionality is influenced by application communication demands and hardware capabilities. This section elaborates on the influence of application characteristics and hardware

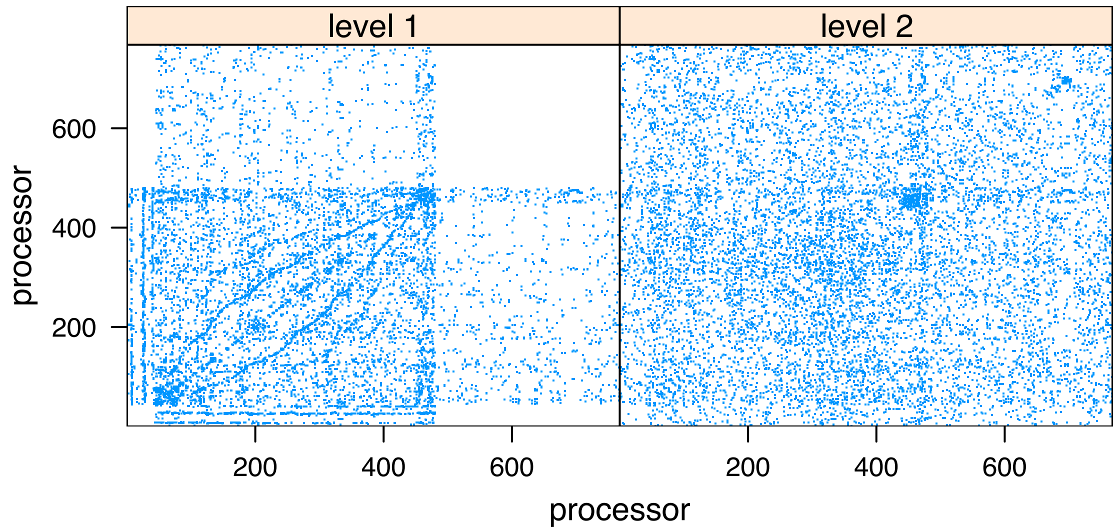


Figure 1.3: Communication characteristics of HyperCLaw showing the message exchange topology during varying mesh refinement.

architecture on library functionality.

1.3.1 Influence of Application Communication Characteristics

Historically, most HPC applications have static communication characteristics. For these applications, communication libraries that provide fixed functionality and support for limited configuration are adequate to serve their communication demands. Most current communication libraries are designed and built based on these demands.

Modern HPC applications, as seen in the previous section, have dynamic communication characteristics; they perform better when protocols in the libraries are changed to match these characteristics. We have seen in the previous section that applications such as AMR and multi-physics codes exhibit these dynamic characteristics. For example, in HyperCLaw's bubble simulating problem, both the number of communicating processes and communicating process pairs varies, as do message characteristics like size and number [12]. For these characteristics, communication libraries that select different protocols based on the number of communicating processes can improve the application performance [13]. Other research [14] has demonstrated that selecting collective algorithms and implementations based on message characteristics can improve the performance of HPC applications.

Further, research has shown that tailoring protocol functionality to an application improves its performance [15]. For example, applications with varying preposting behavior require different protocols to improve their bandwidth. Specifically, applications that prepost most of their large receives can gain substantial benefits from an eager rendezvous protocol; however, eager rendezvous protocols waste substantial network bandwidth when most receives are not preposted. This implies that no one protocol optimization decision is appropriate for all applications.

To better serve the needs of various HPC applications (with static, diverse and dynamic communication needs), in contrast to current libraries which do not change their functionality at runtime, libraries should be designed such that they can tailor their functionality both at compile and run time.

1.3.2 Influence of Hardware Architecture

The hardware architecture of the machine, particularly network architecture, strongly influences the functionality of a communication library. The role of the communication library is to satisfy the communication needs of an application, by taking advantage of network hardware capabilities. For example, if an application requires reliable message transfer, and uses Ethernet, the libraries should add reliability at compile time. If it uses a reliable network such as Infiniband, the libraries should remove reliability at compile time.

Further, research has shown that customized libraries that take advantage of hardware capabilities improve performance characteristics [16]. For example, IBM's customized MPI library for BlueGene/L took advantage of deposit bits and the configurable collective network of the machine. As a result of the customization, the MPI library's collectives provided 75% more bandwidth than a well-engineered generic MPI implementation, MPICH [16]. Similarly HP-MPI [17] used topology information to assign processes to nodes and demonstrated that it improves collective bandwidth.

In HPC machines, as seen in the previous section, network interfaces varies both in architecture and capabilities. Examining the Top 500 list, we observe that 95% of machines use commodity network interfaces, and they choose from as many as 15 different commodity network interfaces. Commodity network hardware, typically, provides only basic transport services. In contrast, proprietary network interfaces provide more capabilites besides basic transport services. For example, Red Storm's Seastar network interface provides network message demultiplexing and remote direct memory access (RDMA) capabilities which are not provided by commodity network interfaces such as Ethernet.

In the presence of network architecture diversity and research showing the utility of functional customization, the communication libraries should be customizable to take advantage of hardware capabilities. Unlike current libraries which have static functionality and limited customization support, libraries should be designed that are easily customizable and change their functionality at runtime.

1.4 Reconfigurable Communication Libraries

1.4.1 Overview

The goal of reconfigurable communication libraries is to improve the application performance by matching protocol functionality to changing communication needs. They must support tailoring their functionality at compile time to match the hardware architecture. They must support changing protocol functionality at runtime to match changing communication demands. They should support selecting algorithms and implementations for collectives at runtime to match network topology and network congestion. They should allow the protocol implementations to choose data structures, and switch data structures based on the runtime state information.

There are many challenges in designing, developing and implementing reconfigurable libraries to improve the application performance. The rest of this section describes these challenges and provides the description of a framework which handles these challenges.

1.4.2 Challenges of Implementing Reconfigurable Communication Libraries

Building reconfigurable libraries that can improve the application performance presents four major challenges:

1. Collecting information about changing application requirements and the runtime state of the system;

2. Analyzing the collected information and converting it to a set of appropriate actions;

3. Reconfiguring the library with little or no disruption of application execution;
4. Supporting all the above functions without increased development and implementation complexity.

Collecting the information

The reconfigurable communication libraries need an application's resources usage characteristics (CPU cycles, memory, network bandwidth) and the hardware capabilities of the system, so they can match their functionality to application requirements and hardware capabilities. The information will help libraries to select appropriate reconfigurations that can improve application performance. For example, to select appropriate algorithms for collectives, the libraries would require network topology and network congestion information.

While collecting this information, the libraries should not disrupt application execution or affect the application performance. Communication information – message characteristics of every point-to-point message, collectives – can grow very quickly with increase in problem size, which can quickly consume memory space, disk space and CPU cycles. Collecting information offline can alleviate some of these problems; however offline information may not always be possible and sometimes information might not be accurate.

Analyzing the information

The information collected has to be converted to a set of reconfiguration actions; the actions are changes to library functionality at a reconfiguration point. The changes can be simply changing control parameters or choosing a different algorithm and implementation for a service. For example, information such as network topology, congestion and message latency may be used to change algorithms for collective operations; resource usage and message characteristics – type and size of message, latency and bandwidth availability, and memory usage – may be used to change message transfer protocols.

The reconfiguration actions may have to be determined from inexact application and environment information. For instance, when information for a large problem size is not available, it has to be inferred from a smaller problem size. When information for certain input data sets is not available, it has to be inferred from application runs with different input data sets. In all these cases, the careful interpretation of approximate information becomes more important.

Ensuring smooth transition of the system after reconfiguration

During reconfiguration, the changes to the libraries in one process can affect one or more processes in the parallel system. Since the functionality changes involve changes to both local and global state of a process, this typically affects more than one process. For example, the changes in local state of a process can affect in-transit data and control messages. It can affect the completion of collectives that are partially completed during reconfiguration.

Reconfigurable libraries should ensure that there is no disruption to the application execution or to the correctness of the solution. They should ensure no state information is lost during a reconfiguration. They should handle in-transit messages, and partially completed operations. They should support co-ordination among processes during a reconfiguration, so processes can make a smooth transition after the reconfiguration. Also, if reconfiguration involves replacing one implementation with another, the libraries should preserve the process state.

Controlling software engineering costs

Reconfigurable libraries should support re-usuability and extension of functionalities. They should support varying levels of function abstractions to implement composition and reconfiguration at varying granularity. Besides extension of the library functionality, the libraries should support adding new reconfigurations and optimizations. These properties would amortize design and development costs of the libraries.

1.4.3 Thesis Statement

A reconfigurable communication framework for incorporating application- and hardware-specific optimizations at compile time and run time can improve performance characteristics of HPC applications.

1.4.4 PRO-MPI

This dissertation describes PRO-MPI, a framework for constructing reconfigurable MPI libraries that can improve performance of HPC applications. The framework enables the libraries to support configuration and reconfiguration. It collects and analyzes the application characteristics required for the reconfiguration. It maps the application characteristics to a set of changes for the libraries that reduces the gap between the application demands and library functionality. It enables the libraries to support re-usuability and extension of their functionality, and addition of new reconfigurations and optimizations.

The main components of the PRO-MPI architecture are the communication layer, profiler, analyzer, and reconfiguration manager. The communication layer provides an MPI library to HPC application which changes its functionality both at compile and run time. The profiler collects application communication characteristics (profiles). The analyzer analyzes the profile and converts it to a set of reconfiguration actions. The reconfiguration manager executes the reconfiguration.

1.5 Contributions of the Dissertation

The main contributions of PRO-MPI described in this dissertation are:

- Demonstration that system software flexibility improves the application performance.
- Design of a framework that helps development and implementation of profile-driven reconfigurable MPI libraries: The framework provides abstractions to collect and understand the application communication characteristics, and proposes change to the protocol functionality (of the communication libraries) to decrease the mismatch between the application communication demands and the library functionality. It also provides the mechanisms for reconfiguration and a structure for the communication layer to support the reconfigurations.
- Evaluation of the framework by implementing two reconfigurable MPI libraries – Open PRO-MPI and Cactus PRO-MPI: Cactus PRO-MPI is a prototype implementation to understand the ability of the framework to implement a reconfigurable MPI and understand the overhead of a reconfigurable MPI. Open PRO-MPI is a reconfigurable MPI implementation that supports full MPI-2 semantics.
- Demonstration of the usefulness of profile-driven reconfiguration: Cactus PRO-MPI and Open PRO-MPI demonstrate the use of past application characteristics to drive changes in the communication libraries and match the protocol functionality to the changing communication demands.

- Evaluation of the reconfigurable MPI libraries by improving the performance of HPC applications and HPC benchmarks: Empirical results demonstrate that Open PRO-MPI can improve performance of HyperCLaw and SAMRAI by assigning the RDMA resources and the RDMA connections based on the application characteristics.
- Design and evaluation of low-cost profile generation methods to drive reconfiguration in high-cost applications.

1.6 Dissertation Outline

This section previews the remaining chapters and describes the overall organization of this dissertation. Chapter 2 compares other frameworks that implement configurable system services and MPI implementations that support reconfiguration and adaptation. Chapter 3 presents an overview of the PRO-MPI framework. Chapters 4 and 5 present details of two reconfigurable MPI libraries – Open PRO-MPI and Cactus PRO-MPI, and show that reconfigurable MPIs improve application and benchmark performance. Finally, Chapter 6 summarizes the conclusions of this dissertation and explores opportunities for future research directions.

Chapter 2

Related Work

The use of profiling, reconfiguration, and adaptation to improve application performance has been an area of active research. Much of this work has dealt with constructing configurable systems for distributed systems, configurable and adaptable transport protocols for wide-area network applications, or supporting very limited reconfiguration in MPI implementations. Very few systems enable full reconfiguration in MPI implementations or provide frameworks for implementing fully functional, reconfigurable communication protocols for HPC applications.

This chapter discusses work related to configurable frameworks and MPI implementations. In Section 2.1, we describe a number of frameworks that provide features for constructing configurable system software. Section 2.2 describes reconfiguration and adaptation support in current MPI implementations. Section 2.3 describes profiling and performance monitoring systems, and highlights differences between their functionality and profiling in PRO-MPI. Section 2.4 describes profile-driven code optimization systems. Finally, section 2.5 summarizes the approaches others have taken in related to the challenges described in chapter 1.

2.1 Configurable Software

Systems that support reconfiguration and adaptation had their origins in distributed systems. Though, PRO-MPI design and goals differ from these systems, there are some inspirations drawn from these systems. This section discusses some of the frameworks that were designed to build configurable system software, and describes the functionality they lack which is useful for building reconfigurable communication libraries for HPC applications.

2.1.1 x-kernel

The x-kernel is a multithreaded operating system kernel, which also provides abstractions for hierarchically composing protocol modules [18, 19, 20]. It provides three abstractions for functional composition: protocol modules, sessions, and messages. Protocol modules implement the operating system and other services' functionality. The sessions are instantiations of the protocols, and messages are used for interaction between the protocol modules. The protocols use session operations, *push*() *and pop*(), for sending and receiving messages between sessions of various levels. The protocol modules at a higher level of hierarchy use services provided by lower level modules using the messages and the interfaces.

2.1.2 Cactus

Cactus is a framework for building configurable services for distributed systems [21]. Like x-kernel, the Cactus supports hierarchal functional composition and provides interfaces for the layered protocols to interact. However unlike x-kernel, the Cactus composition model is a more flexible model for composition. It allows two-level functional composition, and supports more fine-grained functional composition.

In Cactus, a service is composed of protocol layers of composite protocol. A composite protocol is in turn composed of functional modules called microprotocols. A microprotocol is a self-contained software module implemented using an event-based model. They interact with each other using a message abstraction and an event mechanism, and protocol layers interact using the message abstraction.

The Configurable Transport Protocol (CTP) and H-CTP are configurable transport protocols based on the Cactus framework [22, 23]. CTP, a composite protocol, is a transport protocol for wide-area networks. It is composed by choosing microprotocols that define basic transport, fragmentation, congestion control, reliability and flow control. In the current implementation, CTP has the option of choosing from more than one implementation of each type of microprotocol; as a result, CTP can be configured as a protocol that is semantically similar to TCP, UDP, or a mix of both.

H-CTP, which is an optimized version of CTP, is a transport protocol used for grid applications. H-CTP is optimized for performance by reducing overhead of event handling, removing extra message copies, and optimizing the message structure for performance [23].

2.1.3 Appia

Appia is a framework for building communication protocols. In Appia, like Cactus and x-kernel, protocols are layered and composable. These protocols support non-hierarchal composition like Cactus [24, 25, 26]. Their functionality is implemented using an event driven model i.e., the communication functionality is implemented as event handlers, and the handlers are executed when events are raised.

Though Appia provides hierarchal and non-hierarchal composition, its composition model and concurrency model are restrictive when compared to Cactus. Appia's composition model defines QoS, which in turn is defined by a series of protocol layer instances. A QoS definition fails if a layer requests an event and none of the layers provide the event. If QoS succeeds, all events corresponding to a QoS are combined to form a channel. Though channels can share sessions with other channels, it cannot support many of the interaction patterns that are supported by Cactus [25]. Besides the composition model, its concurrency model, which supports one execution thread, is also restrictive when compared to Cactus.

2.1.4 Distrinet Protocol Stack (DiPS)

DiPS is a component framework for constructing self-adapting system software. Its composition model supports both hierarchical and non-hierarchical composition [27, 28]. For implementing composition and adaptation, it provides four abstractions: *Component*, *Message*, *Connector* and *Reflection point*. A *Component* is a self-contained software module which implements well-defined functionality and provides a fixed interface. Components use the *Message* abstraction to interact with each other. A *Connector* is used to connect the components, and to implement a concurrency model. A *Reflection point*, which is used to implement self-adapting behavior, routes the messages based on the information in a *Message*.

2.1.5 Ensemble

Ensemble is a component framework for building customizable protocol stacks. Its protocol stacks are composed of smaller modules called micro-protocol modules [29, 30]. A module provides a specific functionality and interacts with other modules using two fixed event-driven interfaces – upper and lower. They use down events to communicate with lower modules, and up events to communicate with upper modules. Ensemble provides a fixed set of events that modules can choose to define their interaction and functionality. Based on interaction and required functionality, these interfaces can choose events from a fixed set of events provided by the framework.

Ensemble stacks, like DiPS, support coarse-grained protocol reconfiguration both at compile and run time [31]. When an application changes its requirements, Ensemble's detector detects the changes and initiates the protocol switch protocol (PSP). PSP co-ordinates with all the participants in the system to install new protocol stack across all the participants that matches the applications new requirements.

2.2 Reconfiguration and Adaptation in MPI Implementations

2.2.1 Open MPI

Open MPI is an MPI implementation that uses a well-defined component architecture, the MPI Component Architecture (MCA). Communication functionality in MCA is provided by self-contained software modules that support well-defined interfaces. The communication architecture of Open MPI includes MPI, PML (point-to-point management layer), BML (BTL management layer) and BTL (Byte transfer layer) layers. The MPI layer provides the MPI interface, the PML layer provides MPI semantics, the BML layer is responsible for multiplexing the MPI messages, and the BTL layer is responsible for transferring data between network interfaces. More details of Open MPI point-to-point architecture are in 5.2.1, [32], and a detailed discussion of the Open MPI architecture are in [4, 33, 5].

In Open MPI, the support for reconfiguration is limited to a few specific instances. For example, it allows setting up connections between MPI processes' lazily [34]. Besides that, it provides support for functional configuration at compile time.

For example, MCA defines interfaces for implementing different collective communication algorithms and supporting different network interfaces, algorithms and supporting different network interfaces, allowing Open MPI to be customized to run on a wide range of platforms. It does not, however, either support dynamic connection management or dynamic resource management. Also, it does not support selecting algorithms for collective routines based on the application communication characteristics or resource availability.

2.2.2 MPICH

MPICH is a widely used MPI implementation. Its layered architecture maximizes the sharing of hardware independent code and eases code portability across various architectures [35, 6]. The main layer, the abstract device interface (ADI), abstracts hardware dependent code, and decouples the code from hardware independent code. This decoupling eases porting MPICH to a new architecture [36]. There are many implementations of ADI, important among them for MPI is the Channel interface. The Channel interface provides operations and functions for sending messages between any two process spaces.

Like Open MPI, MPICH supports limited adaptation in a few instances. For example, MPIVAPCH, an MPICH adaptation for Infiniband, supports automatic connection establishment [37]. It allows the connections to be established lazily while not allowing dynamic tear down or dynamic re-establishment. Apart from allowing functional configuration in device interface, its ADI allows the user to select a hardware interface from the interfaces that are available at compile time.

2.2.3 Adaptive MPI

Adaptive MPI is an MPI implementation developed using CHARM++, which is an object-based model [38, 39]. It uses CHARM++'s communication, load balancing and threading model. In Adaptive MPI, MPI processes are mapped onto virtual processors. These virtual processors communicate using CHARM++'s message passing objects. As a consequence, MPI process communication is mapped onto CHARM++'s object communication. Similarly, it uses CHARM++'s object migration for MPI process migration.

Like MPICH and Open MPI, Adaptive MPI supports limited reconfiguration and adaptation. Mainly, it supports automatic load balancing [40]. To load balance, it uses CHARM++'s framework and abstractions, particularly its load balancer. The load balancer framework collects processor's workload information, and uses this information to rebalance the unbalanced system by moving MPI process from heavily loaded nodes to lightly loaded nodes. Thus, Adaptive MPI is limited to collecting workload information and rebalancing the workload in the system.

2.2.4 Other MPI Implementations

STAR-MPI is an extension added to MPICH [41, 42], which supports automatic selection of collective algorithms for MPI collective operations. It contains collective routines and multiple routines for each of these routines. During application runtime, when application calls a collective operation, STAR-MPI selects an algorithm and an implementation for the operation based on application characteristics and machine topology.

HP-MPI is an MPI implementation that supports multiple interconnects and multi-protocol execution. Its reconfiguration support is limited to using profile data to make topology aware decisions [17]. For example, during launch of an application, it uses profile data (application data from a prior run) to choose algorithms for collective operations, and place MPI ranks onto the processor cores.

2.3 Profiling and Performance Monitoring

Although, performance monitoring tools and profilers (in this architecture) have a similar functionality (they both collect application data and interpret the data for performance), they both have different goals. A profiler collects application data to understand application behavior, and an analyzer uses this data to suggest protocol functionality changes that could lead to application performance improvement. In contrast, performance monitoring systems collect application data and use this data to identify performance bottlenecks, debug the application, or visualize the profiling results. This section discusses some of the performance monitoring systems, and compares their functionality to PRO-MPI.

Offline performance monitoring tools collect application data during the application execution, and analyze the data after application execution. Traced-based performance monitoring tools such as Paradyn [43] and TAU [44], collect information about the events in a system, and use this data to understand the performance behavior of the system [45, 46]. The trace-based monitoring tools are expensive, as they require analyzing huge amounts of trace data. To reduce this overhead, PHOTON [47] and Supermon [48] use statistical sampling to reduce the trace data before analyzing the data.

Unlike offline monitoring tools, online monitoring tools dynamically insert profiling code into executing programs and dynamically analyze the collected trace data for performance or debugging. The biggest challenge for online monitoring tools is to control the overhead of monitoring and aggregate performance data. The Ganglia monitoring system [49], a distributed monitoring system for HPC systems, controls overhead by using multicast-based listen/announce protocols for monitoring nodes and using tree structures for aggregating the state. Though Ganglia's overhead is low, it uses extra messages for monitoring. Other online monitoring tools based on embedded gossip, reduce the overhead further by not using any extra messages [50, 51]. Online monitoring tools with higher overhead such as Magpie [52] find application in e-commerce systems, where there is a higher tolerance for latencies than in HPC systems.

Both online and offline performance monitoring tools monitor the system and use the data for performance analysis. However, they do not use the collected information to adapt system functionality that could lead to application performance improvement. In contrast, PRO-MPI uses the trace data to change system functionality and improve application performance.

2.4 Profile-driven Code Optimization

Profile-driven code optimization has been used to produce optimized code for a given target architecture. For example, ATLAS (Automatically Tuned Linear Algebra Software) optimizes the code by using the information obtained by a set of pre-run experiments. When the code is first installed on a machine, ATLAS runs a series of experiments to determine the optimal parameters for its kernels. Then it uses these empirical results to tune the kernels so that they are optimized for the processor architecture of the machine [53]. Like ATLAS, FFTW uses profile-driven code optimization to improve the FFT kernel performance [54], and PHiPAC uses a similar technique for optimizing matrix operations [55].

PLTO, a link-time code optimizer, uses profiles of past application runs to drive code optimizations in MPI libraries; particularly it optimizes code by using compilerstyle optimizations such as substituting inline routines, dead-code elimination, and constant propagation [56].

Though both profile-driven code optimizers and PRO-MPI improve application performance by using profile information, there are a few important differences between them. Profile-driven code optimizers improve application performance by changing code which is functionally equivalent to un-optimized code. In contrast, PRO-MPI uses profile-driven reconfiguration to improve application performance by selecting appropriate functionality from the available library functionality. And, PRO-MPI can use these systems to take advantage of optimized code.

2.5 Summary

Many frameworks described in this chapter provide composition models for functional composition. But none of these systems provide mechanisms for building profile-driven reconfigurable systems. They lack mechanisms that can collect application requirements, and convert this information to a set of protocol changes so that the library functionality matches changing application requirements. Also, these systems are not geared towards high-performance computing: abstractions do not support message zero-copy, do not support high-performance network interfaces, does not provide abstractions for building high-performance communication system, do not provide abstractions required for one-sided message semantics, and the composition model overhead degrades application performance.

Unlike MPI implementations mentioned in this chapter, PRO-MPI support for reconfiguration is not limited to specific cases. It provides a framework for constructing fully-functional reconfigurable MPI implementations. The framework provides mechanisms such as a profiler, an analyzer and a reconfiguration manager to collect more wide range application characteristics, and use the profile information to drive a wide range of reconfigurations. It also provides the ability to add new reconfigurations, providing a scope for functional extension and re-use. Further, the framework enables reconfiguration in other MPI implementations that support functional composition.

Chapter 3

PRO-MPI Architecture

3.1 Overview

PRO-MPI is a framework for developing and implementing reconfigurable MPI libraries. Its architecture is designed to ease the implementation of reconfigurable MPI libraries. The architecture does this by enabling the use of profile driven reconfiguration in the libraries.

As discussed in Chapter 1, there are number of challenges in implementing the reconfigurable libraries. To reconfigure and improve the application performance, the libraries have to understand the communication behavior and performance bottlenecks of an application. During and after the reconfiguration, the libraries should not disrupt the application execution, or affect the correctness of the solution. They should support re-usability and extension of their functionality, and support addition of new reconfigurations and optimizations, so that implementing libraries is more viable solution than building and implementing a customized communication library.

PRO-MPI's architecture, shown in Figure 3.1, is designed to address the challenges of developing and implementing reconfigurable libraries. It consists of four main components: communication Layer, profiler, analyzer and reconfiguration manager. The communication layer provides an MPI implementation, which can change its functionality both at compile and run time. The profiler collects information required to understand the application's communication behavior and drive the changes (reconfiguration); it provides this information as a profile to the analyzer. The analyzer uses this information to change the protocol functionality of the communication layer. The reconfiguration manager executes these actions, and ensures smooth transition of the system after reconfiguration.

The remainder of this chapter describes the PRO-MPI architecture in detail. Section 3.2, 3.3, 3.4 and 3.5 describes in detail each of the PRO-MPI components. Section 3.6 provides an example PRO-MPI implementation that improves HyperCLaw performance (an example HPC application described in Chapter 1). Section 3.7 provides a list of potential MPI reconfigurations that could be implemented in the PRO-MPI implementations.

3.2 Communication Layer

3.2.1 Overview

A communication layer provides an MPI implementation and other communication services required by an application. The layer's structure is driven to enable functional configuration and reconfiguration. Its interface hides profiler, analyzer, and reconfiguration mechanisms to provide a standard access to the reconfigurable communication service. Though the communication layer architecture is defined in the context of MPI libraries, it is general enough to construct other communication ser-

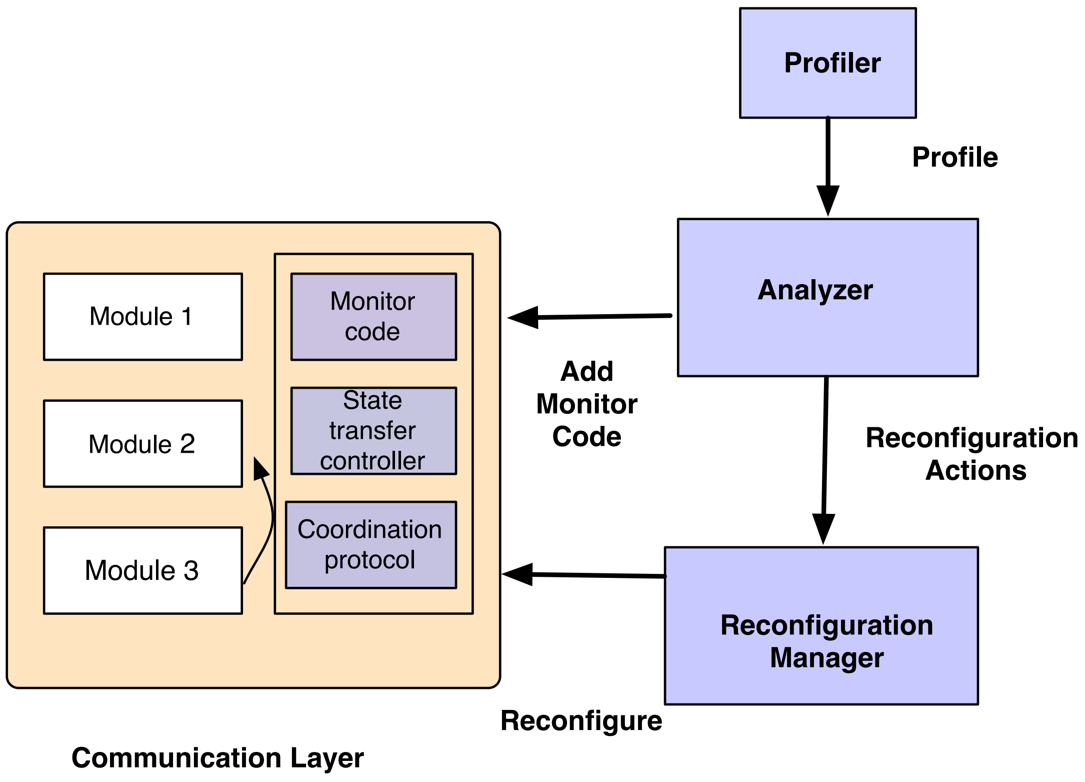


Figure 3.1: PRO-MPI architecture

vices such as OpenMP, TCP or UDP.

The communication layer is a composition of modular functional elements. These modules are self-contained communication functionality implementations, and they provide a fixed interface for accessing its functionality. At compile time, the layer chooses modules to define its functionality based on application characteristics and hardware architecture. At run time, the layer's functionality can be changed by replacing or changing these modules to suit the changing application requirements.

3.2.2 Structure of the Communication Layer

As mentioned previously, the communication layer is a composition of functional modules or other communication layers, which provides a communication service to an application. A module may implement a property for the service, or a different variant of the same property, or some book keeping functionality. For example, a communication layer implementing MPI is composed of modules each implementing a MPI semantic. It might include modules for de-mulitplexing, implementing a small message transfer protocol, and implementing a large message protocol. This structure provides various advantages for implementing a communication service:

- **Configurability:** A communication service when implemented by a communication layer which is a composition of functional modules can be tailored to suit application requirements and hardware architecture.
- **Reconfigurability:** A communication service when implemented as a composition of functional modules that can be added or removed at runtime can be changed at runtime to match changing application demands.
- **Functional Optimization:** As a consequence of support for composition and modularity in the layer's structure, it can choose functional optimizations both at compile time and runtime. Optimization, in a service, is a functional implementation that is efficient for a certain operating scenario. For example, during an MPI application execution, different queue implementations are efficient based on the number of messages queued. If a process' run-time state has a large number of messages, MPI implementations using hash match tables instead of list match tables [1] would reduce traversal latency. On contrast, if a process' run-time state has a small number of messages, MPI implementations

[1] Match table - In an MPI implementation, a match table is a data structure that contains information required for demultiplexing the MPI messages.

using list match tables would conserve memory while keeping the traversal latency low. This structure would allow the communication layer either to choose a hash table implementation or a link list implementation.

- **Reusability:** Modules implementing various protocol functionality can be used in many services. For example, in an MPI implementation, a RTS protocol implemented as a module can be used by large message transfer protocols such as Eager Rendezvous and Rendezvous protocol.

- **Extensibility:** In this structure, adding a new service requires the developer to implement only missing functional modules in the library. For example, a UDP-like transport protocol can be changed to a reliable message transfer protocol by implementing modules that add sequence numbers, provide acknowledgement for a message, and retransmit the lost message.

3.3 Profiler

3.3.1 Overview

A profiler collects application characteristics and operating environment information to understand the application behavior and drive reconfiguration. For example, to choose a collective algorithm and implementation for MPI_Allreduce during runtime, the profiler would collect the number of times the application uses MPI_Allreduce, topology of the machine, latency of the messages, and congestion information in the network.

For a parallel application, the profiler collects application characteristics for each process, which are then combined into a global profile. During a parallel application execution, each process might run on a different physical node. A local profile represents application characteristics of a process, which is a snapshot of its communication and resource usage characteristics. During runtime, local profiles are generated, which are then combined at the end of application execution to generate a global profile.

The key challenge for a profiler is to control the overhead (both computation and memory) of collecting a profile. It typically collects message characteristics and operating environment information of an application in all its phases. While collecting operating environment information, the local profile size usually stays constant with increasing problem size. However while collecting the message characteristic information, the local profile grows exponentially as the number of messages increases with increasing problem size. The architecture controls this overhead by filtering out all messages that are irrelevant for a reconfiguration.

3.3.2 Types of Profiles

As mentioned above, a profile is a snapshot of an application's communication characteristics. In profile-driven reconfiguration, an application run during which a profile is collected is called the *profile run*, and an application run during which the communication layer functionality is changed to match application communication demands is called the *reconfiguration run*. Based on the information contained in the profile and timing of the profile collection, we have four different classes of profiles.

- **Exact and Inexact Profiles**
  Profiles are collected in a profile run, and are used to drive a reconfiguration in a reconfiguration run. The operating conditions – input data sets and input parameters for the application, problem size, system size and target hardware – between these runs can remain the same or change. If they remain the same, the profiles used to drive a reconfiguration are *exact profiles*. If operating

conditions are different for the profile run and the reconfiguration run, since the profiles contain only partial information to drive the reconfiguration, they are called *inexact profiles.*

- **Online and Offline Profiles**

  The profiles that are generated during an application run and saved to the disk, which are then used to drive a reconfiguration in a different application run are offline profiles. In contrast, online profiles are generated and used in the same application run.

  Online profiling is computationally expensive, and affects the application performance adversely [57]. In this architecture, to limit the profiling overhead, all online profiles are only local profiles i.e., an analyzer uses only local online profiles to recommend changes to library functionality.

3.4 Analyzer

An analyzer recommends changes to the communication layer functionality based on a profile and also schedules when changes will take effect. The goal of these changes and their timing is to match communication layer functionality to changing application requirements. This section elaborates more on the functionality and different types of analyzers.

3.4.1 Role of the Analyzer

Mapping Application Characteristics

The main functionality of an analyzer is to map profile information to communication layer functionality changes that could improve application performance. It receives the application profile from the profiler, which is a snapshot of an application's communication characteristics. The profile may have whole or partial information to drive these changes. In this architecture, the analyzer is in agreement with the profiler on the profile representation i.e., analyzer understand the format of a profile.

The functionality changes that the analyzer proposes can be a change in protocol functionality, parameter value or an algorithm or implementation for an operation. For example, the analyzer might select algorithms and implementation for the collectives based on a profile that contains network topology, message latency and congestion information. Sometimes the profile has only partial information; in that case, the analyzer uses numerical analysis tools to construct the missing information. For example, when a profile from a smaller problem size is used to drive a reconfiguration in a larger problem, the analyzer interpolates the partial profile to construct a complete profile, which is then used for the analysis.

Monitoring System State and Operating Environment

In addition to the mapping, the analyzer triggers a reconfiguration in the layer by monitoring a layer's internal state and external operating environment. Typically, it monitors message characteristics (messages exchanged, collectives, message buffers used), and system state such as queue lengths, memory used, message latency, bandwidth and congestion in the network. When a certain system state is reached, the reconfiguration manager is informed to execute the reconfiguration. The analyzer provides all the information needed to execute the reconfiguration.

Monitor code, in addition to triggering the reconfiguration, is also responsible for detecting the application phases – a specific synchronization point of an application, or phase in the application execution after which communication characteristics change. The application phase detection depends on the hints provided by an application (this architecture at present does not support automatic generation of reconfiguration points). Using the hints, state information, and the schedule of changes, the monitor code decides a change in application phase. In all reconfigurations mentioned in this dissertation (chapter 4 and 5), we use an application phase as a reconfiguration point.

3.4.2 Online and Offline Analyzer

The analyzer maps profile information to a set of protocol functionality changes. This mapping can be done offline i.e., when the profile run and reconfiguration run are different, or it can be done online i.e., when the profile run and reconfiguration run are the same.

In offline analysis, the analyzer provides protocol functionality changes to a reconfiguration manager before the start of a reconfiguration run. To recommend these changes, the analyzer needs the snapshot of applications communication characteristics in a comprehensible format. In this architecture, for offline analysis, the profiler provides a global offline profile which is obtained from a previous application run. Before the start of execution of an application, the analyzer uses the profile to decide the changes in protocol functionality for each application phase.

In online analysis, the analyzer provides protocol functionality changes to a reconfiguration manager only before the start of an application phase. This requires both online profiling and online monitoring. This architecture as mentioned in the previous section supports generating online local profiles. The local profile provides a snapshot of communication characteristics of an application's previous communication phases. For online monitoring, the analyzer uses the monitor code to learn system state, resource usage, and network conditions of previous application phase. During start of each application phase, the analyzer uses the local profile and system state information to decide the changes in protocol functionality for that phase.

3.4.3 Analysis of Inexact Profiles

Inexact profiles have partial application communication characteristics information to drive a reconfiguration during a reconfiguration run. For analysis and to drive a reconfiguration, the complete information on application characteristics is required. To construct this missing information, the analyzer uses numerical analysis methods such as interpolation and extrapolation [58]. While using these methods, the partial information is used for some processes which are determined by the numerical methods, and for the remaining processes their information is derived or approximated from the partial information.

3.4.4 Overhead of a Reconfiguration

Each reconfiguration action proposed by the analyzer requires computing, network, and memory resources. If the change is either a change in protocol functionality or replacement of a module, the overhead includes the cost of transferring the state and quiescing all the process. An application experiences this overhead as an increase in message latency, as a decrease in bandwidth, the unavailability of the CPU, or all of these.

The analyzer proposes the change to the protocol functionality after a cost-benefit analysis of the reconfiguration. It considers, for cost-benefit analysis, the number of times a change is required, resources required for each change, and performance benefits of the change. For example, the analyzer proposes a change in data structure for the prepost receive queue from a link list implementation to a hash implementation, if the change provides a total reduction in message latency. At each reconfiguration point, the change of implementation for the queue is allowed by the analyzer if total sum of reduction in message traversal latency for all messages during the application phase is greater than overhead of state transfer (message pointer and book keeping information), and the delay in processing receives, and demultiplexing messages.

The cost of a reconfiguration is specific to the reconfiguration, we provide the details of cost-benefit analysis done by the analyzer for each reconfiguration as we encounter them in this dissertation.

3.5 Reconfiguration Manager

The reconfiguration manager provides mechanisms for executing a reconfiguration, and mechanisms for ensuring smooth transition of a system after reconfiguration. This section presents architecture details of these mechanisms.

3.5.1 Protocols for System Operation

To ensure a smooth operation of the system after reconfiguration, the reconfiguration manager provides two protocols: a state transfer protocol and a co-ordination protocol.

- **State transfer protocol**: The goal of this protocol is to transfer state information from the older module to a newer module. The reconfiguration manager uses a simple reliable packet transfer protocol as a state transfer protocol. The older module acts as a server, and the newer module acts as a client during state transfer. When replacing one communication layer by newer communication, each module in a communication layer acts as a client and a designated leader module acts as a server, which synchronizes the state data into global state data. The global state data is then transfered to a leader in the new communication layer, which is responsible for distributing the state to all its modules.

Before the state transfer, the modules are required to convert the state information into flat data packets. The newer module converts the flat data packets into appropriate state information. The state transfer protocol does not impose constraints on packing formats; the decision is left to individual reconfiguration implementations.

- **Coordination protocol**: The goal of a coordination protocol is to synchronize all processes for a reconfiguration. The need for co-ordination arises from the fact that in a parallel application more than one independent process is involved in reaching a solution, and the changes in one process affect the global state. For example, if for a process, one message transfer protocol implementation is replaced by an alternate implementation, this reconfiguration affects all processes receiving the message and it also affects in-transit data and control messages. This protocol, in such a reconfiguration, would ensure all processes interpret the received message correctly and no in-transit data or control messages are lost.

  The co-ordination protocol implementation involves two parts, one that synchronizes the processes, and another part that quiesces all processes. The synchronization is implemented using a barrier. When a process hits a barrier, it waits for all other process to reach this point. At this point, the system moves to a quiesce state where inter-process communication is delayed, in-transit data and control messages are temporarily queued, and data to the application is temporarily queued. As a result in the quiesce state, the process library is safe to reconfigure.

3.5.2 Mechanism for Implementing a Reconfiguration

The goal of this mechanism is to implement a reconfiguration. It is a control flow system that on receiving a trigger for changing the functionality, uses state transfer and coordination protocol to implement the changes.

Reconfiguration manager receives a trigger from an application or monitor code to make changes to the protocol functionality. The application might trigger the reconfiguration indicating the change in application phase (communication characteristics). The monitor code can also trigger the reconfiguration indicating the change in resource availability or operating conditions.

After being triggered, the reconfiguration manager uses the coordination protocol to force the communication libraries to reach a synchronized state. Each process executes a barrier routine (as part of coordination protocol) provided by the reconfigurable library to reach the synchronized state. All processes in the system wait at the barrier and switch to a quiescent state. In quiescent state, all processes access only library functionality that is not changed by a reconfiguration.

In addition to the trigger event, the reconfiguration manager also receives the schedule of changes to the communication library. Based on these changes, the reconfiguration manager marks the modules and the communication layers for update. If the changes involve replacing a module, the modules engage the state transfer protocol. If the changes does not involve replacement, then modules are simply updated with new functionality.

3.6 Example PRO-MPI Reconfiguration to Improve HyperCLaw Performance

Figure 3.2 shows a PRO-MPI reconfiguration for HyperCLaw. As described in Chapter 1, HyperCLaw exhibits dynamic communication characteristics – both the number of communicating processes and processes pairs taking part in a communication vary between time steps. Managing resources and using protocols based on these characteristics can improve HyperCLaw performance.

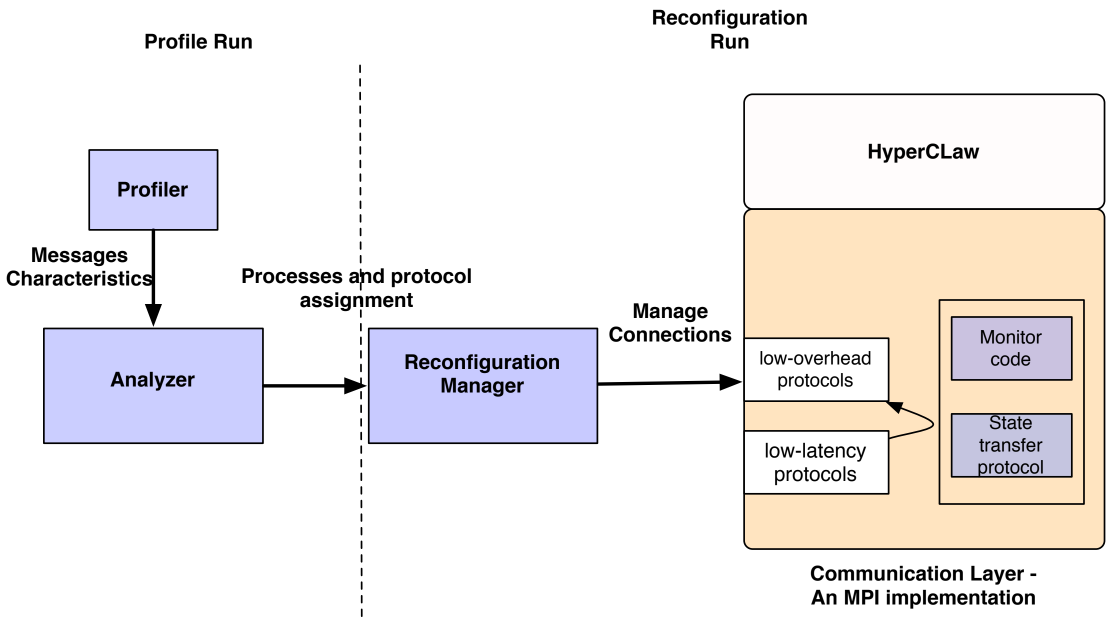


Figure 3.2: An example PRO-MPI framework for HyperCLaw

PRO-MPI can improve HyperCLaw performance by efficiently using low-latency connections. As research shows, the overhead of managing low-latency connections can degrade application performance when large numbers of processes use them [34]. So, current MPI implementations use a limited number of low-latency connections. They assign these limited connections statically to processes that communicate during the initial phase. For applications such as HyperCLaw, which has dynamic communication chracteristics, this static assignment underutilizes the connections. To improve the utilization, PRO-MPI dynamically assigns a limited number of connections to process pairs based on messages exchanged and communication topology. Thus dynamic assignment improves the application performance by assigning low-latency connections to process pairs that exchange large number of messages.

To carry out this reconfiguration, each component plays a role as shown in the Figure 3.2.

- **Communication layer**, which is an MPI implementation, supports both configuration and reconfiguration of its functionality. It would allow selecting protocol functionality at compile time based on the network hardware architecture. It would allow assigning low-latency protocols and high-bandwidth protocols based on application message characteristics.

- **Profiler** collects HyperCLaw message characteristics during the profile run. These message characteristics include message size, number of messages sent by each process, and source and destination of each message. It also collects resources used by each MPI process, particularly registered buffers that are used by low-latency connections.

  To collect message characteristics, the MPI libraries are instrumented with profiler code. Besides message characteristics, the profiler code also logs application phase information, registered buffer usage and low-latency connections established by each process. The log information (local profile) is saved to a flat text file on the disk. It is then combined, at the end of the application execution, with other local profiles into a global communication profile of the application.

- **Analyzer:** During the profile run, the analyzer converts the profile – message log of each process – into a set of process pair priorities. It prioritizes each process pair based on message exchange frequency. Since the message exchange frequencies change after each application phase (for the class of applications considered here), the priority of process pairs might also change after each application phase.

These process priorities determine the processes that are allowed to open low-latency connections. For example, if low-latency connections are limited to n (n < number of application processes), the analyzer assigns low-latency connections to n process with the lowest priority. The analyzer then sends this information to the reconfiguration manager, which uses it for assigning connections during a reconfiguration run.

- **Reconfiguration manager:**

  After being triggered at the end of an application phase, during the reconfiguration run, the reconfiguration manager modifies the low-latency connection allocation for all processes. At the end of an application phase, it first safely resets all low-latency connections, and then uses the information provided by the analyzer to make new connection assignments. Thus, the reconfiguration manager performs dynamic connection assignment based on message exchange frequency and message exchange topology.

3.7 MPI Reconfigurations

Table 3.2 lists MPI optimizations which could be implemented in a PRO-MPI implementation. For each optimization, a table entry provides a description of the optimization, MPI routine (operation) that is improved, message characteristic that is relevant for the optimization, performance characteristic that is improved, operating conditions that is required, and cost of the optimization. Table 3.1 provides the meanings of some terms used while describing the optimizations.

To reconfigure, PRO-MPI chooses an optimization based the message characteristics of an application and operating conditions. As seen in the table, each MPI operation has many optimizations each for a different message characteristic or a different operating condition. The analyzer based on the application profile would recommend an appropriate optimization (that improves a performance characteristic) for the operation.

Table 3.1: List of terms that are used to describe the optimizations

Term	**Description**
Process	*A MPI process*
Sender	*A MPI process that sends a MPI message.*
Receiver	*A MPI process that receives a MPI message by posting a matching receive.*
Matching receive	*A receive posted on a receiver that has matching bits (source, destination, tag) which matches a particular message.*
Control data	*MPI header information*
Small message	*A MPI message which has data that is less than 12 KB; however in most implementations this a configurable value.*
Large message	*A MPI message which has data that is greater than 12 KB; however in most implementations this a configurable value.*
RDMA	*A user-level or hardware-level message transfer protocol which allows the sender to write the message into the receiver's application buffer.*
Short queues	*For these optimizations, it is assumed that queues of less than 50 items are short queues.*

Long queues	*For these optimizations, it is assumed that queues of greater than 50 items are long queues.*
MPI_Send (standard)	*This notation is used to indicate a standard mode MPI_Send. In this mode, the MPI implementation can decide to buffer the outgoing message or not buffer the message. If it buffers the message, the send completes once the message is buffered, and irrespective of its matching receive status on the receiver. If the message is not buffered, the send is completed once the matching receive is posted on the receiver, and the message is copied to the application buffers.*
MPI_Send (synchronized)	*This notation is used to indicate a synchronized mode MPI_Send. In this mode, the send can be started whether or not a matching was posted. However, the send is completed after the receive is posted, and the transfer of message has started.*
MPI_Send (buffered)	*This notation is used to indicate a buffered mode MPI_Send. In this mode, the send can be started whether or not a matching receive was posted, and the operation can be completed before a matching receive is posted.*

Table 3.2: List of MPI optimizations that could be implemented in a PRO-MPI implementation

Description of the Optimization	**Improved MPI Operation**	**Message Characteristics**	**Improved Performance Characteristic**	**Optimal Operating Conditions**	**Cost**
Eager: Send matching data and control (matching) information eagerly	*MPI_Send (standard, synchronized)*	*Small messages*	*Reduces message transfer latency for small messages*	*Reduces latency when a matching receive is posted*	*When a matching receive corresponding to the send is not posted, buffer space is wasted on the receiver process*
Rendezvous: Send control (matching) information and RTS to the receiver, and send data after receiving a CTS	*MPI_Send (standard, synchronized, buffered)*	*Long messages*	*Improves message bandwidth*	*Improves bandwidth when a matching receive is not posted.*	*When a matching receive corresponding is not posted, eager rendezvous protocol provides better message bandwidth*
Eager Rendezvous: Send matching (control) information and data to the receiver irrespective of posted receive status	*MPI_Send (standard, synchronized, buffered)*	*Long messages*	*Improves effective message bandwidth*	*Improves bandwidth when a matching receive is posted*	*When a matching receive corresponding to the send is not posted, the data has be sent twice*
RDMA mechanism with persistent buffers: Send data from an user application buffer to a receiver's application buffer	*MPI_Send (standard, synchronized, buffered)*	*Small messages*	*Reduces message transfer latency for small messages*	*Reduces small message latency, when few processes are communicating*	*Polling overhead affects scalability and performance, when a large number of processes are communicating*

Send/Receive: Send data to a receiver library and copy the data from the library memory to the application memory buffers	*MPI_Send (standard, synchronized, buffered)*	*Small messages*	*Reduces message transfer latency for small messages*	*Reduces small message latency as send/receive does not have the polling and registration overhead like RDMA mechanism*	*Copy overhead (from library buffers to application buffers) adds to the message latency*
Preposted receive queue, unexpected message queue, and send queue implemented as hash tables	*MPI_Send MPI_Recv*	*Large number of unmatched messages and/or large number of posted receives*	*Reduces queue traversal latency and thus message latency*	*Reduces queue traversal latency for long queues because of the hash queue's constant look up time*	*For short queues, the hash function evaluation and memory allocation overhead can affect the message latency*
Preposted receive queue, unexpected message queue, and send queue implemented as linked lists	*MPI_Send MPI_Recv*	*Small number of unmatched messages and/or small number of posted receives*	*Reduces queue traversal latency and thus message latency*	*Reduces queue traversal latency for short queues*	*Queue traversal latency increases linearly with queue length and hence increases the message latency for long queues*
Buffering: A unmatched message on the receiver is buffered until a matching receive is posted	*MPI_Send MPI_Recv*	*Large number of preposted receives*	*Improves achieved application bandwidth*	*More effective usage of the network bandwidth*	*If a matching receive is posted after long delay after the message arrives, scalability suffers as memory requirement increases*
Pulling: A unmatched message is dropped on arrival at the receiver, if a matching receive is not posted by the receiver, and data has to be sent again by the sender	*MPI_Send MPI_Recv*	*Small number of prepost receives*	*Improves achieved application bandwidth*	*More effective use of memory, if only a small percentage of receives are posted*	*If a matching receive is posted after the message arrives, the message is dropped wasting bandwidth and CPU resources*

Application bypass for reduction operations: Reduction operations are performed on a group of processes organised as a tree. With application by-pass, the parent process need not wait while the children processes perform operations on the data and send it to the parent process.	*MPI_Reduce (all reduce operations)*	*Process skew*	*Improves CPU utilization*	*In presence of process skew, the parent process has to wait for the children processes. However with application by-pass, this CPU time can be utilized for other computations*	
Matching queues implemented as associative list on hardware	*MPI_Send, MPI_Receive*	*Large number of prepost receives*	*Reduces queue traversal latency and thus reduces message transfer latency*	*Improves queue traversal latency for long queues because of constant look up time*	*For short queues lengths (< 70) shows no improvement*
Ready-mode Receive: Post a receive without searching in the un-matched receive queue. This type of receive can be used in correspondence with ready-mode send as ready-mode does not start until a matching receive is posted	*MPI_RSend*	*Messages sent using ready mode send*	*Reduces queue traversal latency and thus reduces message transfer latency*	*Proves to be advantageous when there is a long un-matched queue*	*Not advantageous for short un-matched queues*
MPI fence immediate method: Use two barriers for synchronizing the processes in a communicator during the one-sided operations.	*MPI_Win_Fence*	*When most or all processes of a communicator participate in a MPI_Put (Get, Accumulate)*	*Reduces synchronization overhead for one-sided MPI operations*	*Reduces synchronization overhead for machines with a hardware implemented barrier and a low latency network*	$Cost = 2 * (log\ p) * lat$; *Where p is the number of process in a communicator; lat - latency of a message*

MPI fence deferred method: Use one reduce-scatter operation for synchronizing the processes of a communicator during the one-sided operations.	*MPI_Win_Fence*	*When most or all processes of a communicator participate in a MPI_Put (Get, Accumulate)*	*Reduces synchronization overhead for one-sided MPI operations*	*Cost has one latency term, so this method is preferable over Immediate method for a machine with high latency network*	$Cost = (log\ p) * lat + 2(p-1) * (t)$; *lat - latency of a message, t - transfer time for a byte*
Post-Start-Complete-Wait immediate method: Wait for messages from all the processes in a group during a MPI_Start, MPI_Complete and MPI_Wait	*MPI_Post (-Start-Complete-Wait)*	*When only few processes of a communicator participate in a MPI_Put (Get, Accumulate) operation*	*Reduces synchronization overhead for one-sided MPI operations*		*Cost = 2*g*lat ; g-number of processes in a origin and target group; lat - latency of a message*
Post-Start-Complete-Wait deferred method: Waits for messages from all the processes in a group only for MPI_Win_Complete	*MPI_Win_Post (-Start-Complete-Wait)*	*When only few processes of a communicator participate in a MPI_Put (Get, Accumulate) operation*	*Reduces synchronization overhead for one-sided MPI operations*		*Cost =g*lat ; g-number of processes in a origin and target group ;lat - latency of a message.*
Lock-Unlock with immediate synchronization	*MPI_Lock (Unlock)*	*When only two processes participate in a MPI_Put (Get, Accumulate) operation*	*Reduces synchronization overhead for one-sided MPI operations*		*Cost = 4*latency of a message*
Lock-Unlock with immediate method synchronization	*MPI_Lock (Unlock)*	*When only two processes participate in a MPI_Put (Get, Accumulate)*	*Reduces synchronization overhead for one-sided MPI operations*	*If only one remote operation is performed or atleast one MPI_Get operation is performed*	*Cost < 4*latency of a message transfer*

RDMA pipeline protocol: Overlap RDMA operation and memory registration upto pipeline depth, and free the registrations after the send operation	*MPI_Send (standard, synchronized, buffered, ready-mode)*	*Large messages*	*Decreases memory requirements over RDMA Direct protocol*	*Applications with lower buffer reuse*	*With higher buffer reuse, RDMA Direct protocol is more advantageous*
RDMA Direct protocol: Message transfer using registered buffers that are cached for future reuse	*MPI_Send (standard, synchronized, buffered, ready-mode)*	*Large messages*	*Amortizes memory registration overhead and increases bandwidth*	*Applications with higher buffer reuse*	*With lower buffer reuse, RDMA Pipeline is more advantageous*
RDMA message transfer only on static memory	*MPI_Send (standard, synchronized, buffered, ready-mode)*	*Small messages*	*Eliminates memory registration overhead and reduces message transfer latency*	*When processes use a small number of small messages*	*RDMA operations are restricted to static memory and requires copy-in copy-out if more memory is required*
RDMA message transfer using on the fly memory registration	*MPI_Send (standard, synchronized, buffered, ready-mode)*	*Small messages*	*Reduces message transfer latency*	*When processes use a small number of small messages*	*Registering message at message transfer increases message latency*
Scalable Fault tolerant protocol for parallel runtime environment based on k-ary sibling tree	*Parallel runtimes such as Open MPI's runtime environment, mpd of mpich, Harness of FT-MPI*		*Improves scalability and fault tolerance of a parallel run time*	*Improves fault tolerance as it can tolerate upto k process failures (for network to be bisected)*	

3.8 Summary

This chapter described architectural details of the framework, PRO-MPI, for constructing reconfiguration MPI libraries. It provides architectural details of four main components of the framework: profiler, communication layer, analyzer and reconfiguration manager. Also, it explains the role of each component and interactions of these components during a reconfiguration with an example.

PRO-MPI architecture provides a set of abstractions for implementing reconfigurable MPI libraries. The profiler and analyzer together understand the communication behavior of an application, and propose a set of changes to the protocol functionality that can improve the application performance. The mechanisms to implement the changes are decoupled into another component of the framework: the reconfiguration manager. And the communication functionality is decoupled from all other components.

Chapter 4

Cactus PRO-MPI Implementation

4.1 Overview

The architecture described in Chapter 3 provides a framework (PRO-MPI) for constructing reconfiguration MPI libraries. This chapter presents a reconfigurable MPI implementation, Cactus PRO-MPI, based on the framework. The prototype implementation also includes an example reconfiguration. We demonstrate that this reconfiguration can improve synthetic benchmark performance.

In this chapter, we present details of the Cactus PRO-MPI implementation, which is based on Cactus and the Configurable Transport Protocol (CTP). Cactus PRO-MPI is implemented as an enhancement to CTP. At present, it only supports point-to-point semantics of the MPI specification. To support reconfiguration, the implementation includes the profiler, analyzer and reconfiguration manager. The profiler, analyzer and reconfiguration manager are designed to functionally support preposted receives reconfiguration.

The remainder of the chapter describes this implementation. Section 4.2 describes Cactus and CTP. Section 4.3 describes the details of Cactus PRO-MPI implementation. Section 4.4 describes the example reconfiguration implemented in the prototype. In Section 4.5, we provide an evaluation of the implementation using the reconfiguration, and measure the performance impact on the benchmarks.

4.2 Cactus and CTP

Cactus PRO-MPI is an enhancement to the CTP, which in turn is implemented in the Cactus composite protocol framework. This section briefly describes Cactus and CTP. A detailed description of Cactus and its execution structure can be found in [59], of the message abstraction used by Cactus in [60], and of CTP itself in [61]. The Cactus description is mostly extracted from our paper [15].

4.2.1 Cactus

Cactus is a system for constructing highly-configurable protocols for networked and distributed systems. It supports two-level functional composition, which provides flexibility required for building the configurable services, particularly, network protocols. Individual protocols in Cactus, generally termed *composite protocols*, implement a protocol functionality such as transport protocol or RPC. They compose hierarchically to provide a configurable service. They are inturn constructed from fine-grained software modules called *microprotocols* that interact using an event-driven execution paradigm. Each microprotocol implements a different function of the protocol. Cactus, in addition to functional composition, provides a message abstraction that is optimized for configurable systems.

Two-level functional composition (composite protocol and microprotocol) is designed to provide flexibility and functional granularity for building highly configurable protocols. Composite protocols include protocol functions such as push, pop and demux operations, which are responsible for moving the data in and out of the protocol. Besides the protocol functions, the composite protocol includes microprotocols each of which implements a protocol function or property, and data abstraction that is shared by all the micro-protocols. For example, if a composite protocol implements a transport protocol, the microprotocols in the composite protocol might add sequence numbers to messages, segment messages, or provide flow control. A microprotocol implementation includes initialization code, private data, and a collection of event handlers. They interact with each other using an event driven model.

Processing of structured messages by microprotocol-defined event handlers comprises the basic programming model of Cactus. Events are used to signify state changes of interest, such as "message arrival from the network". When such an event occurs, all event handlers bound to that event are executed. Events can be raised explicitly by microprotocol instances or implicitly by the composite protocol runtime system.

The Cactus runtime system provides a variety of operations for managing composite protocols, microprotocols, events, and event handlers. Composite protocols and microprotocols are created by initializing protocol objects using *cInitProtocol()* and *cInitMicroProtocol()*. Microprotocols are added to the composite protocol using *cAddMicroProtocol()*. Microprotocols can create new events, in addition to default events, and delete events using *cCreateEvent()* and *cDeleteEvent()*. In addition to traditional blocking events, Cactus events can also be raised with a specified delay to implement time-driven execution, and can be raised asynchronously. Other operations are available for unbinding handlers, halting event execution, and canceling a delayed event.

Finally, for building configurable protocols, Cactus supports a message abstraction, named *message attributes*, and a synchronization mechanism. In this abstraction, data is added to the message as a named attribute. Named message attributes enable message customization that is required for configurable systems. A microprotocol adds custom information using named message attributes in a message without affecting other microprotocol's information. The scope of the named attributes correspond to composite protocol (local), a single node (stack), or to all the nodes the message travels (network). Cactus runtime provides *SetMsgAttr(message, scope, attribute, data)*, and *GetMsgAttr(message, scope, attribute)* operations to set and get the attribute values.

Synchronization and coordination of execution activities in Cactus is accomplished through *hold bits* that may be associated with data items. The hold bits are a set of boolean flags associated with a message. These hold bits are used mainly to coordinate activities such as sending and deallocating a message across multiple microprotocols. Cactus provides two types of hold bits in each message to control these activities, send bits and deallocate bits. Each microprotocol that needs to control the message is allocated a hold bit. When all send bits in a message are set, the message is allowed to move up/down the protocol stack. And, when all deallocate bits are set, the message is deleted.

4.2.2 CTP

CTP is a message-oriented configurable transport protocol written in the Cactus framework, primarily for use on local-area and wide-area network (e.g. Ethernet) connections. CTP includes a wide range of microprotocols for operating in this environment, including microprotocols implementing acknowledgements (`PositiveAck`), retransmissions (`Retransmit`), forward error correction (`ForwardErrorCorrection`), and a range of congestion control mechanisms and policies (`WindowedFlowControl`, `TCPCongestionControl`, etc.). Using these and other microprotocols, researchers have implemented CTP configurations that support TCP-like, UDP-like, and SCTP-like semantics.

Microprotocols in CTP handle a set of predefined events, particularly those that indicate message availability from the network or an application. Two primary events are used for processing outgoing messages - `MessageFromUser` indicates that a new arbitrary-sized message is available for transmission, while `SegmentToNet` events are generated by fragmentation/reassembly microprotocols that fragment messages into segments for transmission over the network. Similarly, receive-side processing includes `SegmentFromNet` and `MessageFromNet` events, which again correspond to fragmented packets and reassembled messages. Each microprotocol can bind these handlers and set message attributes as appropriate.

CTP uses Cactus' hold bits for coordinating the message activities. It uses send bits for sending the message to an application or on to the network, and deallocate bits for controlling the life span of a message. For example, the congestion control and flow control microprotocols use send bits to control the message going to the network, and ordering microprotocols controls the message going to the application using send bits. Each microprotocol that processes the message is allocated a send bit and deallocate bit. The hold bits allows the microprotocol to process the message independent of other microprotocols.

4.3 Cactus PRO-MPI Infrastructure

To prototype PRO-MPI, we add a profiler, an analyzer, and a reconfiguration manager to CTP. This section presents implementation details of these components and interactions between them. It also describes extensions to CTP protocol functionality to provide MPI semantics and support reconfiguration.

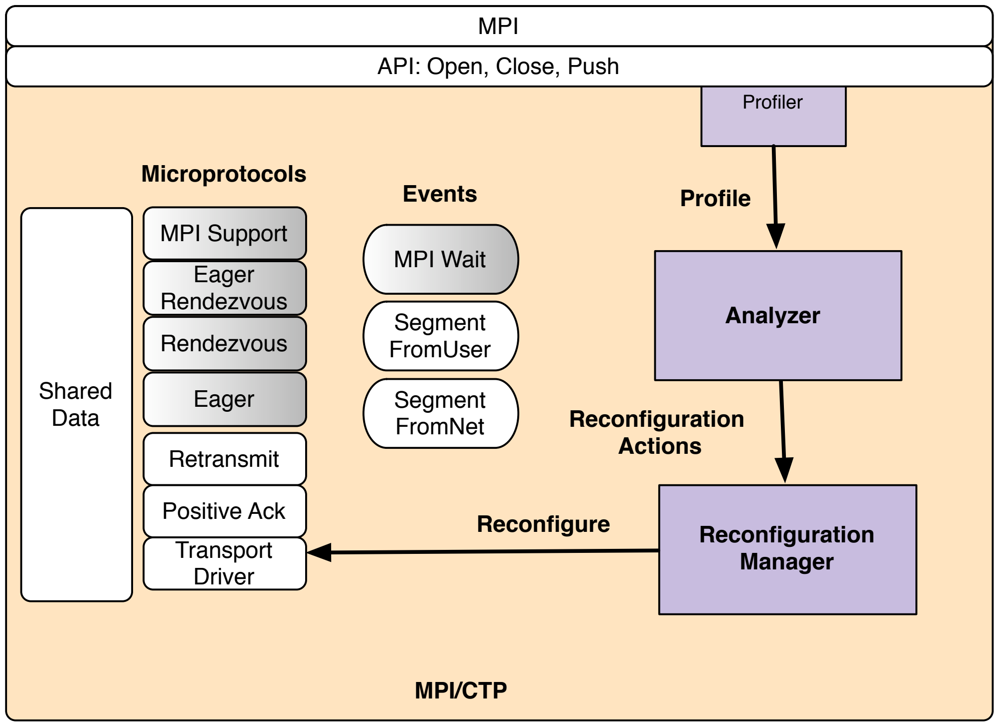


Figure 4.1: Architecture of Cactus PRO-MPI showing the profiler, the analyzer, and newly added microprotocols and events. The newly added microprotocols and the events are shown as grey colored components.

4.3.1 Communication Layer (MPI/CTP)

Our implementation of MPI/CTP includes a variety of additions, particularly new microprotocols, new message attributes, and careful interaction with existing CTP microprotocols. The new microprotocols implement different MPI-specific protocol algorithms and the new message attributes are used to carry MPI-specific information for these microprotocols. The following sections describe these extensions. A diagram illustrating these changes is shown in figure 4.1.

MPI Support Microprotocol. Because CTP originally used TCP-like message demultiplexing based on port numbers instead of MPI matching semantics, we first had to introduce protocols that customized CTP to support MPI matching semantics. The `MPISupport` microprotocol is responsible for implementing basic MPI matching semantics in CTP by receiving post requests from the applications through the CTP control interface and making posted and unexpected queues available to other microprotocols for their use. An API to these lists is provided for other MPI/CTP microprotocols to use as necessary. In addition, `MPISupport` handles miscellaneous local requests that do not require message generation and processing, for example calls to `MPI_Wait()`. Note that `MPISupport` introduces a new `MPIWait` event to CTP to signal threads blocked on synchronous MPI calls.

Message-Handling Microprotocols. MPI/CTP includes message-handling microprotocols for sending MPI messages over the network. MPI/CTP currently includes 3 microprotocols related to sending and receiving MPI-oriented messages: `Eager`, `Rendezvous`, and `EagerRendezvous`; these correspond to common techniques for sending short (`Eager`) and long (`Rendezvous/ EagerRendezvous`) MPI messages.

Like most CTP microprotocols, each microprotocol implements handlers for the `SegmentFromUser` and `SegmentFromNet` events to enable them to process messages. In response to messages from the application to send, these microprotocols may send the message immediately or send a `RTS` to facilitate later transmission. Likewise, in response to `SegmentFromNet` events, they may do nothing and rely on preexisting CTP microprotocols to handle acknowledgements, or they may send or schedule transmission of data to the requester if the received packet is an `RTS` or `CTS`.

The MPI/CTP microprotocols also set send and deallocate bits to coordinate message transmission and deallocation with other CTP microprotocols, and set message attributes to transmit control information. We have added a handful of new message attributes for the MPI-specific microprotocols, particularly `RTS/CTS`, rank, tag, and communicator fields.

Interactions with Existing Microprotocols. By writing MPI functionality as an extension to CTP, MPI/CTP configurations retain full access to other CTP microprotocols that provide functionality that may be desirable in some cases. For example, the `PositiveAck` microprotocol can be used to acknowledge message receipt in a short-message protocol without having to reimplement and reoptimize acknowledgement functionality. Similarly, microprotocols such as `Retransmit` and `WindowedFlowControl` allow MPI/CTP protocol configurations to work seamlessley in long-haul and lossy networks. Because all such functionality in CTP is optional, MPI/CTP configurations running over standard high-speed reliable fabrics (e.g. Myrinet) need not pay the price for this functionality.

4.3.2 Profiler and Analyzer

The profiler and the analyzer are added to the CTP to collect message characteristics and understand application communication behavior.

The profiler, in particular, collects message and resource usage characteristics. The MPI layer (MPI layer shown in the Figure 4.1) is instrumented with the profiler code that parses the MPI message header to record the message size, type, source and destination of each message. To collect information about the MPI protocols used for the message transfer, the event handlers and microprotocols are instrumented with profiler code that records the event handlers and microprotocols executed for each message. Also, the preposted receive queues and wait message queues are instrumented with the profile code that records the lengths of queues at prescribed intervals. The profiler uses the recorded information to generate a profile, which is interpreted by an analyzer.

The analyzer maps the profile into a set of reconfiguration actions. It has an offline component and an online component. The offline component analyzes the profile and generates a set of protocol changes - change control parameter values, enable or disable event handlers, and enable or disable microprotocols. The online component, mainly the monitor code, is implemented as part of the communication layer. It is responsible for monitoring the state of the system (communication layer), and triggering the reconfiguration. In Section 4.4, we demonstrate with an example how the analyzer monitors the state of the communication layer and triggers the reconfiguration.

4.3.3 Reconfiguration Manager

The reconfiguration manager can modify MPI/CTP functionality in three ways. First, it controls the parameter values passed to the event handlers. Second, it can disable and enable event handler execution, when an event is raised. Third, it can enable and disable a microprotocol at runtime. The support for functionality modification at multiple granularity provides enough flexibility for many reconfigurations.

All these changes are implemented by the *MPISupport* microprotocol. To remove a microprotocol, it unbinds all event handlers corresponding to the microprotocol using *UnbindHandler()* operation. And to add a microprotocol, it binds all event handlers corresponding to the microprotocol using *cBindHandler()* operation. In next section, we demonstrate with an example the functionality of the reconfiguration manager.

4.4 Preposted Receives Reconfiguration

To improve the bandwidth availability of MPI applications and benchmarks that have varying message arrival rates and that prepost receives at varying rates, we implemented a preposted receives reconfiguration. The above mentioned dynamic behavior of an application results in varying MPI queue lengths, particularly preposted receive queues. The reconfiguration improves the application bandwidth by enabling the sender MPI process to choose the transport protocols based on queue length information.

In the implementation, the profiler collects queue lengths and corresponding bandwidth information (while using both `Rendezvous` and `EagerRendezvous`). The analyzer converts the information into a schedule which recommends the protocols to be used for various preposted receive percentages. At runtime, the analyzer's monitor code monitors the queue lengths, and triggers reconfiguration based on the previously generated schedule. On receiving the trigger, the reconfiguration manager at the sender process makes the protocol switch between `Rendezvous` and `EagerRendezvous` or vice versa.

The reconfiguration manager implements the reconfiguration (protocol switch) by a combination of microprotocol reconfiguration and filtering code in message-passing microprotocols. Only those message-passing microprotocols that are configured into a given MPI/CTP configuration (and hence have bound appropriate event handlers) can process a message, allowing different message-transmission algorithms to be configured and reconfigured at a coarse scale. MPI/CTP uses this level of protocol switch, namely reconfiguration, between microprotocols that process similar messages, for example between the `Rendezvous` and `EagerRendezvous` message-processing microprotocols.

More fine-grained protocol adaptation on a message-by-message basis, specifically the message size-based protocol switch, is done by parameterization. In particular, each message-passing protocol is designated as either a long-message or short-message protocol, a global shared variable that designates the switch-point between long and short messages is exported by `MPISupport`, and each configured microprotocol only handles an outgoing message if it is of the appropriate size. Note that this requires that only one short and one long message protocol be configured into MPI/CTP at a given time.

4.5 Evaluation

To evaluate our Cactus PRO-MPI design, we used a simple prototype implementation of the MPI point-to-point calls using the design described in Section 4.3. This implementation runs on Myricom GM, supports all of the various MPI point-to-point calls, but does not support MPI collective communications.

We tested two different elements of our Cactus PRO-MPI prototype, namely basic message-passing bandwidths and message-passing bandwidth with reconfiguration. We compared message-passing bandwidth of Cactus PRO-MPI with Open MPI and MPICH (both are production-quality implementations), to understand the overhead of Cactus PRO-MPI. To understand Cactus PRO-MPI's reconfiguration ability, we measured message-passing bandwidth for fixed-size messages with protocol reconfiguration based on the percentage of messages preposted at the receiver.

4.5.1 Setup

We tested the above mentioned scenarios between two dual-processor 2.2 GHz Pentium III Xeon machines with Myrinet Lanai7 adapters [62]. Each machine ran Linux kernel version 2.4.2 and GM 2.1.1 [63, 64, 65]. We compared bandwidths of our implementation versus those of Open MPI 1.0.2 and MPICH/GM 1.2.6.

To run the above scenarios, we used a synthetic benchmark which varied the posted receives on the receiver MPI process while a sender MPI process sends the messages. Using this benchmark, we measured bandwidth between the processes keeping the percentage of posted receives constant. We measured Cactus PRO-MPI and the other two MPI implementations bandwidth using benchmarks from The Ohio State University [66].

4.5.2 Synthetic Benchmark Results

Cactus PRO-MPI Overhead

Figure 4.2 shows the basic bandwidth performance of our prototype implementation. As can be seen in Figure 4.2, our prototype achieves approximately 81% of the point-to-point bandwidth of the OpenMPI or MPICH/GM implementations. The performance difference is due to the costs of extra copies that the existing CTP framework currently imposes on our prototype. Eliminating these copies should make Cactus PRO-MPI bandwidth-competitive with OpenMPI.

Figure 4.3 shows how Cactus PRO-MPI bandwidth varies by percentage of preposted receives with 32 KB messages. As can be seen, the standard rendezvous protocol outperforms an eager rendezvous protocol when 80% or less of receives are preposted. For carefully written applications where most receives are preposted, it is well known that an eager large-message protocol can acheive better performance [67]. This effect can be easily seen in MPI/CTP.

MPI/CTP Protocol Reconfiguration

To test the ability of Cactus PRO-MPI to optimize MPI behaviour through dynamic protocol reconfiguration, we enabled the reconfiguration manager to reconfigure which long message protocol Cactus PRO-MPI used based on feedback from the analyzer on the average percentage of receives preposted at the receiver. The

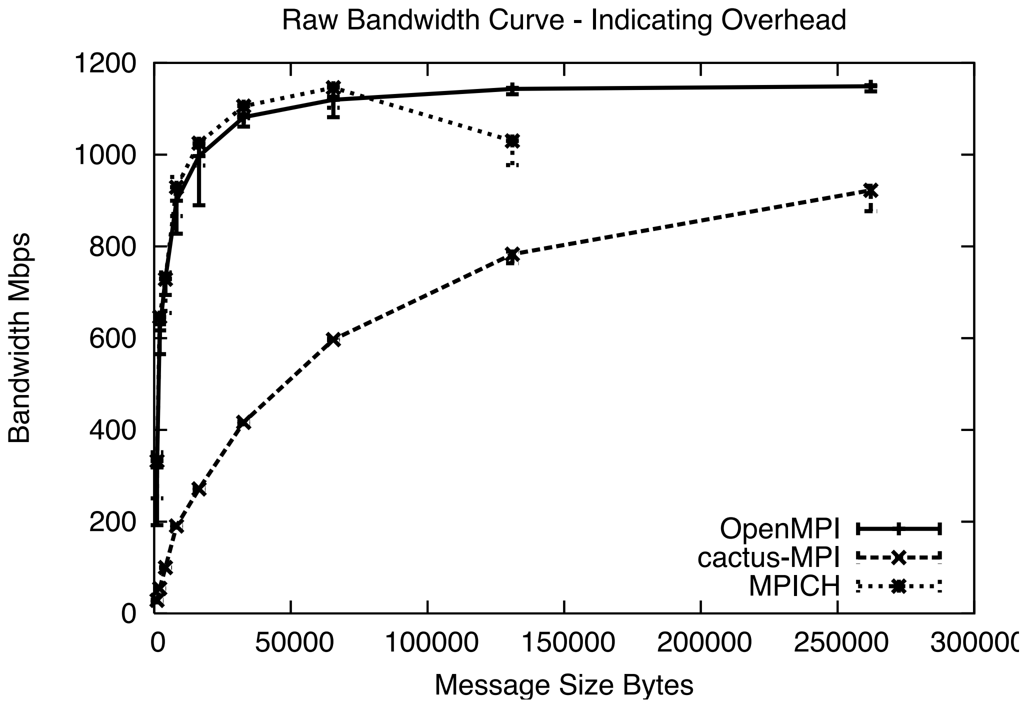


Figure 4.2: Comparative Eager Message Bandwidth

reconfiguration manager then dynamically changed between the `EagerRendezvous` and `Rendezvous` long message protocols by binding and unbinding handlers in each microprotocol at runtime; cutoffs for the protocol switch were determined ahead of time based on the information shown in Figure 4.4.

Figure 4.4 shows that reconfiguration in Cactus PRO-MPI allows it to dynamically adjust its behavior based on remote application behavior, thereby optimizing available MPI protocol bandwidth. As Cactus PRO-MPI becomes more carefully tuned, we expect this to allow applications to acheive better MPI performance by dynamically reconfiguring protocol behavior based on application needs.

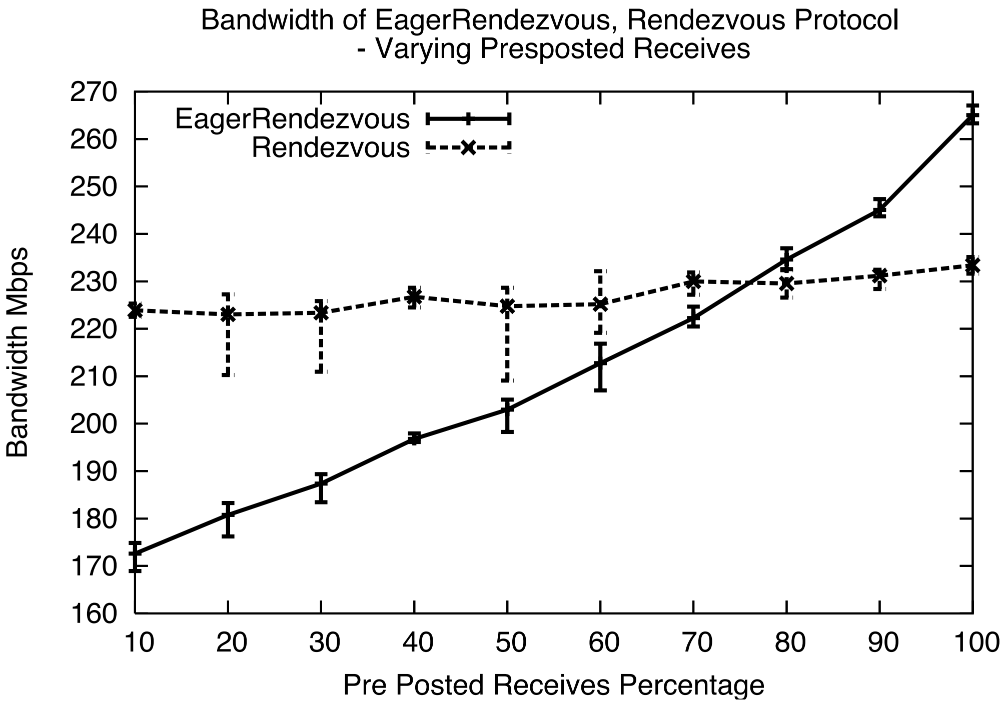


Figure 4.3: Bandwidth By Protocol

4.6 Summary

In this chapter, we evaluated the design of PRO-MPI with an example implementation Cactus PRO-MPI. Cactus PRO-MPI demonstrated the feasibility of the PRO-MPI architecture for building reconfigurable systems. Using the prototype, we showed how reconfigurable MPI implementation can deal with dynamic application behavior, for example changing percentages of preposted receives, and to reconfigure at runtime based on this changing behavior. It demonstrated the usefulness of flexibility at various granularity levels (microprotocols, event handlers and composite protocol) to reconfigure functionality. The results showed the Cactus PRO-MPI's ability to reconfigure with very little overhead. However comparing Cactus PRO-MPIś bandwidth with Open MPI and MPICH shows that it can achieve only 80%

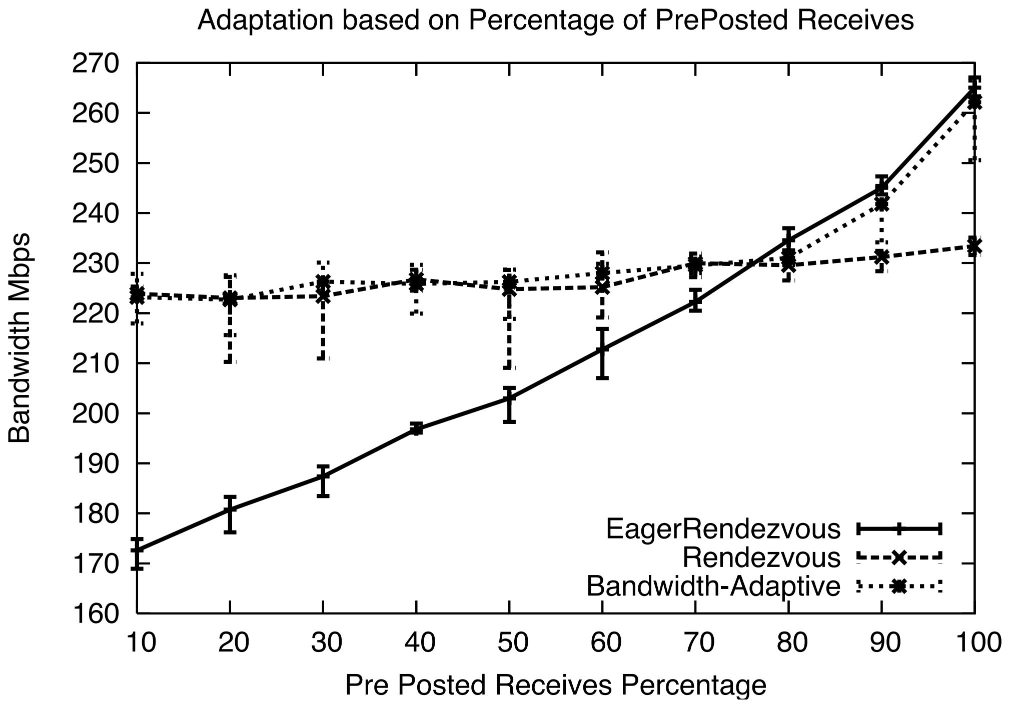


Figure 4.4: Reconfigurable versus Fixed Protocol Bandwidth

bandwidth achieved by Open MPI or MPICH. This overhead as mentioned in Section 4.5.2 is due to the CTP and Cactus implementation.

Chapter 5

Open PRO-MPI Implementation

5.1 Overview

In this chapter, we present a reconfigurable MPI implementation, Open PRO-MPI, based on PRO-MPI and Open MPI that supports reconfiguration and full MPI semantics. The implementation includes two reconfiguration implementations both of which, as we will show, improve the performance of HPC applications.

Open PRO-MPI is implemented as an extension to Open MPI by adding PRO-MPI components – profiler, analyzer, and reconfiguration manager. Open MPI acts as a communication layer that provides an MPI implementation to the applications. The profiler, analyzer, and reconfiguration manager modify Open MPI's protocol functionality in response to changing application characteristics. Open PRO-MPI components are geared particularly towards optimizing RDMA connections and registered buffer usage in Open MPI.

The remainder of the chapter describes this implementation and an evaluation of its performance. Section 5.2 describes Open MPI in which reconfigurations were implemented. It also describes Infiniband and various protocols it supports for message transfer. Section 5.3 describes other components of Open PRO-MPI. Section 5.4 describes the example reconfigurations implemented in Open PRO-MPI. In Section 5.5, we provide a evaluation of the implementation using these reconfigurations, and measure the performance impact on HPC applications and benchmarks.

5.2 Open MPI and Infiniband

5.2.1 Open MPI

Open MPI is a MPI-2 implementation [1], which was designed and implemented after drawing experiences from other MPI implementations such as LAM/MPI [68], LA-MPI [69], and FT-MPI [70]. Besides being MPI-2 complaint, Open MPI supports concurrent and multi-threaded applications. Its design goal is to efficiently support various hardware architectures such as supercomputers, clusters of commodity nodes, and grids (nodes on a wide-area network). To efficiently support various architectures, Open MPI supports message transfer on various network interfaces which includes TCP/IP, Myrinet [71, 62], Quadrics [72], and Infiniband; it utilizes available multiple network interfaces to maximize bandwidth availability by network striping and improve fault tolerance by dynamically handling the loss of network devices.

Open MPI uses a well-defined component architecture, the MPI Component Architecture (MCA) [5]. The component architecture contains three functional areas: MCA, component frameworks, and modules. MCA manages and provides services to the component framework such as passing parameters from a higher layer to components and modules, and configuring, installing and initializing components. Each component framework performs a particular task in Open MPI. For example, Open MPI includes component frameworks such as a point-to-point transport layer and

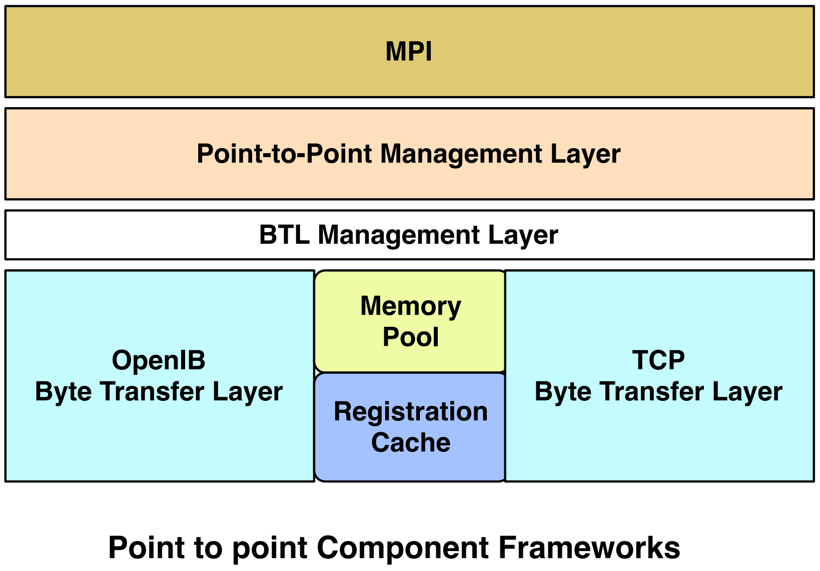


Figure 5.1: Open MPI point to point component architecture

a point-to-point management layer for managing and implementing point-to-point operations. The component frameworks are also responsible for managing modules.

Modules are self-contained software components that implement communication functionality. Modules, which needs to be included, are specified in the Open MPI configuration scripts; during Open MPI compilation, scripts discover the modules and these modules are compiled with other Open MPI code. They can be included in component frameworks either as static libraries or shared libraries; this flexibility provides the ability for including third-party modules that do not provide module source code. The included modules can choose to be enabled at runtime. If enabled, they are initialized and allocated resources.

To support point-to-point communication over specific networking hardware, Open MPI uses mainly three component frameworks: PML (point-to-point management layer), BML (BTL management layer), and BTL (Byte transfer layer). Figure 5.1 provides a general architecture of Open MPI's point-to-point implementation.

PML provides various MPI point-to-point semantics: Standard, Buffered, Ready and Synchronous. It supports various MPI protocols (Eager send and Rendezvous), and is capable of using various types of interconnects (send/recv and RDMA). Besides that, the PML provides higher-level services such as message fragmentation and reassembly and matching necessary to support the MPI interface over a given BTL layer. BTL abstracts network interfaces, such as Ethernet, Infiniband, Myrinet, to provide data transfer services; it implements send/recv operations and RDMA operations(GET and PUT) over the network interfaces. The BML layer is a thin management layer between PML and BTL, which is mainly responsible for multiplexing MPI messages across BTLs. Besides PML, BML and BTL, the point-to-point communication architecture uses the Mpool component for memory management and the Rcache component for caching memory registrations.

Our work has focused primarily on optimizing the use of the OpenIB BTL, a BTL component that transfers data over Infiniband links using OpenIB's verbs interface. It is managed by the OB1 and DR PML framework. OB1 and DR PML implement MPI protocols, buffer messages and use the OpenIB BTL for message transfer on Infiniband interfaces. Each port on the Inifiniband interface has a module initialized for it. When using OB1 and OpenIB BTL, short messages are sent using an eager protocol and large messages are sent using a rendezvous protocol. Since Infiniband interfaces are RDMA capable, eager messages can use either send/recv protocol or RDMA protocol based on resource availability. Long messages, based on the flag settings, can use send/recv, RDMA Direct protocol or RDMA Pipeline protocol.

5.2.2 Infiniband

The Infiniband specification was originally proposed as a general I/O technology [73]. However, currently, it is mainly used for interprocess communication in many modern HPC machines. It provides various features to support communication in HPC machines. Particularly, it supports RDMA and OS bypass required for low-latency and high-bandwidth message transfer [74, 75].

Open MPI's BTL, OpenIB BTL, which uses Infiniband fabric for message transfer, provides various RDMA protocols. For large message transfer, it provides RDMA Direct and RDMA Pipeline, and for small message transfer, it provides a eager RDMA protocol. The long message protocols are optimized for controlling the cost of memory buffer registerations. In RDMA Direct protocol, the memory registrations are cached for the lifetime of an application execution, amortizing the registration cost. RDMA Direct protocols require the buffers to pre-registered, if they are not registered on first use, however after the first use, the buffers are registered and cached for subsequent use. In the RDMA Pipeline protocol, the buffers are registered during message transfer and de-registered after message transfer. However, the buffer registration cost is masked by transferring the message using send/recv protocol during registration. Since the RDMA Pipeline protocol does not cache buffer registrations for future use, the buffers are re-registered every time an application reuses the buffers. More details of these protocols are in [34].

5.3 PRO-MPI: Open MPI infrastructure

To support profile-based protocol reconfiguration and optimization, PRO-MPI supplements Open MPI with three additional elements. Figure 5.2 is a logical representation of Open PRO-MPI, which is implemented as an extension to Open MPI.

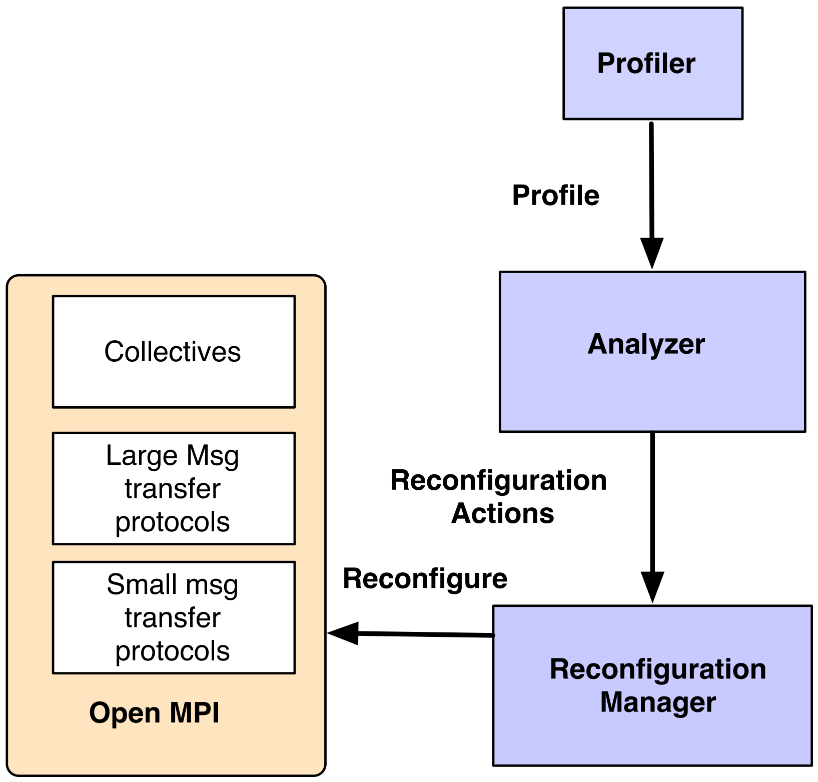


Figure 5.2: Open PRO-MPI components and their use to control a protocol reconfiguration.

In this implementation, the profiler collects information about application communication characteristics into a communication profile for later use; the analyzer uses globally collected profiling information to generate schedules to drive reconfiguration in later application runs; and the reconfiguration manager dynamically changes communication settings based on previously generated schedules. This section discusses details of Open PRO-MPI components, and their integration with Open MPI.

5.3.1 Profiler

The profiler generates a communication profile by collecting application characteristics and operating environment information. The information collected is geared to drive a reconfiguration in a latter application run. Typically a profile has message characteristics - size, type, frequency, resource usage information, network information - message latency, network congestion, and hardware architecture characteristics. In this implementation, the function of profiler is limited to collecting message characteristics, and resource allocation and usage characteristics of an MPI application.

To generate an MPI application communication profile, the profiler collects and combines MPI message characteristics, hardware dependent message characteristics, and buffer usage characteristics. Profiler code in OB1 (PML component) logs every MPI message to determine MPI protocol (eager or rendezvous) used for message transfer, MPI semantics (standard, buffered, ready, synchronous) of the message transfer, source and destination processes, size and type of the message. And, the profiler collects hardware dependent message characteristics – transfer protocols (send/recv or RDMA), message fragment latency – by logging every message after message fragmentation at the OpenIB BTL component. To collect buffer usage characteristics, the profiler uses wrapper code around Mpool and Rcache components that collects buffers used, frequency of reuse and lifetime of the buffers. At every reconfiguration point, this profile is combined with other process profiles to generate a global profile.

5.3.2 Analyzer

The analyzer uses an application profile to generate schedules to drive two reconfigurations: optimizing RDMA connection usage and optimizing registered buffer usage. To optimize RDMA connection and registered buffer usage, it provides a set of process pairs for each reconfiguration point that maximizes the use of RDMA connections and registered buffers. Since the process pairs that exchange the most messages are expected to maximize the use of RDMA connection and buffers, it selects process pairs based on their priority, which is dependent on the number of messages exchanged between them.

The analyzer scripts filters all relevant messages for a reconfiguration from the profile. It uses messages attributes to determine the messages exchanged between any two processes. To optimize RDMA connections based on the message exchange frequency, the analyzer filters all MPI small messages. Algorithm 1 shows the analyzer's code fragment that filters the small messages from an application profile. The input to the code fragment is the application profile, and the output is a message matrix that contains messages exchanged between any two processes in all application time steps. In the code fragment, the function $MESSAGES()$ is used for extracting a message and its attributes from the profile, the functions $SOURCE()$ and $DESTINATION()$ are used to extract the message attributes such as the source and the destination of the message.

The process pairs are prioritized based on the messages exchanged between them. Algorithm 2 shows the code fragment that assigns the priorities to the process pairs. These priorities fill a $PriorityMatrix$ (a 3 dimensional matrix), which is then accesible to the reconfiguration manager. An element, p[i][j[k], of the $PriorityMatrix$ is a priority between process i and j at time step k (a reconfiguration point). In the code fragment, the function $SORT\_DESENDING()$ sorts a array in the descending order, and the function $PRIORITY(x, y)$ provides the priority of the element x in the array y.

The analyzer does additional processing if the operating environment of a reconfiguration run differs from the profiling run. If the problem size of a reconfiguration run is different from a profiled application run, the analyzer extrapolates

Algorithm 1 Pseudocode to determine the number of small messages exchanged between any two processes from an application profile.

```
Logfile ← profile
$MaxTimeStep$ ← input
for $timestep = 1$ to $MAXTimeStep$ do
  while $MESSAGES(Logfile, timestep)$ do
    message ← $MESSAGES(Logfile, timestep)$
    if $SIZE(message) \leq 12KB$ then
      $src$ ← $SOURCE(message)$
      $dst$ ← $DESTINATION(message)$
      $MessagesExchanged[timestep][src][dst] + 1$
    end if
  end while
end for
```

the 3-dimensional matrix to match the problem size of a reconfiguration run. The analyzer can also handle other changes in operating environment such as change in application input parameters.

5.3.3 Communication Layer and Dynamic Resource Management

As seen in the Figure 5.2, we use Open MPI as the communication layer for Open PRO-MPI. It provides an MPI implementation to the applications. As mentioned in Section 5.2.1, we focused mainly on modifying OpenIB BTL to support reconfiguration. To achieve this, we changed the implementation to add a notion of reconfiguration points, support dynamic resource allocation and dynamic connection

Algorithm 2 Pseudocode to compute the priorities of connections between any two processes.

```
$MessagesExchanged \leftarrow$ input
$MaxTimeStep \leftarrow$ input
**for** $timestep = 1$ to $MaxTimeStep$ **do**
  **for** $process_i = 1$ to $MaxProcesses$ **do**
    $TempMessage \leftarrow MessagesExchange[timestep][src]$
    $TempMessage \leftarrow SORT\_DESENDING(TempMessage)$
    **for** $process_j = 1$ to $MaxProcesses$ **do**
      $connection_{i,j} \leftarrow MessagesExchanged[timestep][process_i][process_j]$
      $priority \leftarrow PRIORITY(connection_{i,j}, TempMessage)$
      $PriorityMatrix[timestep][process_i][process_j] \leftarrow priority$
    **end for**
  **end for**
**end for**
```

management, and in this section we discuss these changes in detail.

Reconfiguration Points

At a reconfiguration point, the protocol functionality changes in response to changing application characteristics or operating environment; at a reconfiguration point, all processes of the application reach a global consensus on protocol changes, and changes takes effect without affecting the correctness of the application execution. A reconfiguration point can be an application time step or the end of a particular application phase (change in communication characteristics, change in resource usage pattern, change from computation phase to communication phase or vice versa).

Open PRO-MPI uses a counter in the PML layer to keep track of reconfiguration points. Since the counter is implemented in PML, it is accessible to the applications, and other components and modules of the communication layer. MPI applications access the variable by using non-state changing MPI calls (for example, we use *MPI_Comm_get_name*). Other modules and components in Open PRO-MPI can modify the variable using newly-introducted internal functions such as $get\_reconfigpoint(), set\_reconfigpoint()$ and $inc\_reconfigpoint()$; as the names indicate, $get\_reconfigpoint()$ returns the present value, $set\_reconfigpoint()$ changes the value, and $inc\_reconfigpoint()$ increments the reconfiguration counter.

Dynamic Resource Allocation

To be addressable and act as a communication endpoint in Infiniband, each process requires many resources which includes QP (queue pair) and their corresponding work queues (send work queue, receive queue) and completion queues. OpenIB BTL allocates all these resources during component and module initialization. Besides these resources, Infiniband also requires another important resource – registered buffers – which are used for sending and receving the messages bypassing the OS. Registered buffers are valuable resources, as physical pages that are used by registered buffers cannot be reclaimed for other processes. Therefore registering all required buffers during initialization should be avoided.

Buffer registration during runtime is an expensive operation and it has to be controlled in all time critical paths including message send and receive. Open MPI provides two protocols for managing registrations costs during message transfer: RDMA Direct protocol and RDMA Pipeline protocol. More details of the protocols are in Section 5.2.2.

Unfortunately, both these message protocols are insufficient for many applications. For example, if the RDMA Direct protocol is used for applications with spatial characteristics – where buffer reuse is very rare – caching of memory registrations can cause resource exhaustion, and using the RDMA Pipeline protocol for these applications would have adverse performance effects. Using either protocol for applications with both temporal and spatial characteristics [1] can have adverse performance effect and result in inefficient resource usage. This makes both optimizations ineffective for application with dynamic characteristics and applications with spatial characteristics.

Open PRO-MPI combines approaches of RDMA Direct and RDMA Pipeline protocol providing a effect of dynamic resource allocation. In Open PRO-MPI, before a RDMA write or read operation, if memory is not registered, it is registered using Open MPI's Mpool component (Mpool component provides memory management functions such as allocate, deallocate, register and deregister). And the memory registration is cached using Open MPI's Rcache component; Rcache provides memory registration caching functions such as find, insert and delete. These cache entries are deleted and memory is de-registered after each reconfiguration point by the reconfiguration manager. As a result, all memory is available for future use by the application or for other operating system operations. This approach also avoids the resource exhaustion caused by the RDMA Direct protocol, and overhead of using send/recv protocol in RDMA Pipeline protocol. Besides that, when only limited registered buffers are available, Open PRO-MPI increases registered-memory utilization by allowing only selected processes to use the RDMA Direct protocol. The selection is based on probability of buffer reuse. The details of this process selection are explained in Section 5.4

[1] An application with higher memory buffer reuse is said to have temporal characteristics, and if an application reuses the memory buffers rarely then it is said to have spatial characteristics.

Dynamic Connection Management

Open MPI transfers small message either using the send/recv or the RDMA protocol. To support RDMA operations in Infiniband, the receive buffers need to be registered before a message is sent. A receiver process learns message completion by polling receive buffers. To limit polling time and registered buffer usage, Open MPI uses both the send/recv and the RDMA protocols for small message transfer. The RDMA connections are lazily assigned to process pairs after they communicate and exchange messages up to a certain threshold. A configuration variable controls and limits the number of RDMA connections. After the limit, the process cannot initiate a new RDMA connection, and any small message transfer for a process that does not have a RDMA connection uses the send/recv protocol.

In Open PRO-MPI, RDMA connections are established and reset at every reconfiguration point. The RDMA connections that can be initiated by a process are limited and controlled by a configurable variable. All RDMA connections between process pairs are established at a reconfiguration point. And, at the end of the reconfiguration point, the reconfiguration manager resets the connections, and transfers the state information associated with the process to a new RDMA connection or it is deleted if there is no new RDMA connection. Before the reset, the reconfiguration manager sends the pending message fragments. For the next reconfiguration point, the reconfiguration manager selects the process pairs and initiates the connections. The selection process is explained in Section 5.4). Like Open MPI, in Open PRO-MPI, processes use the send/recv protocol to send message to other processes without a RDMA connection.

5.4 Example Reconfigurations

Using Open PRO-MPI, we have implemented two protocol reconfigurations: connection-protocol reconfiguration and registered-buffer reconfiguration. This section provides implementation details of these reconfigurations.

5.4.1 Connection-protocol Reconfiguration

Connection-protocol reconfiguration is designed to improve the performance of applications that dynamically change how many and with which peers they communicate frequently at runtime. Specifically, we have implemented connection-protocol adaptation between one-sided and two-sided communication protocols at the Open MPI BTL layer based on profiles gathered from previous application runs. In this reconfiguration, PRO-MPI profiler gathers information about how often a process communicates with each peer during application-indicated phases. We also added a one line call to each application to indicate to PRO-MPI when a new application phase begins (phases could also potentially be inferred from application behavior, for example calls to `MPI_barrier`).

In subsequent runs on related (but not necessarily identical) inputs, the reconfiguration manager controls which BTL communication protocol is used to communicate with each peer in each application phase using a schedule generated offline by the analyzer from the profiling information. This allows PRO-MPI to dynamically control which peers use RDMA channels, improving overall application performance as we will describe in Section 5.5.

The profiling information and analysis needed for this reconfiguration is relatively simple. Specifically, the profiler gathers the number of small messages (less than 12 KB) sent to each peer by each process during each indicated application phase. The analyzer uses the profile information to generate a *priority matrix* as well as the maximum number of RDMA connections allowed for each process ($maxRDMA$). The three-dimensional priority matrix contains the communication priority between any two MPI processes during each application phase; specifically element $p[i][j][k]$ of the priority matrix contains the priority of communication from process i to j at application phase k.

The reconfiguration manager initiates, resets, and manages RDMA connections based on data in the priority matrix generated by the analyzer from profiling information. At a given application time step, two MPI processes p (the initiator) and q may be connected via RDMA, if the priority of that connection is higher than other MPI processes, and the current number of RDMA connections p is less than *maxRDMA*. Figure 5.3 shows how connections could be reconfigured between between MPI process i and other MPI processes over an application run when *maxRDMA* $= 2$.

The reconfiguration manager also controls how the communication protocol of a MPI process is switched between RDMA and Send/Receive. At each reconfiguration point, all RDMA connections change state from CONNECTED to QUIESCE. In the QUIESCE state, RDMA connections cannot be used for any new message transfers. After all pending messages are transferred, the resources allocated to the connection are freed and the RDMA connection state is changed to DISCONNECTED. The connection state for a process is changed back to CONNECTED only after the process establishes a new RDMA connection to another MPI process.

5.4.2 Registered-buffer Reconfiguration

The goal of this reconfiguration is to improve application performance by choosing an appropriate large message RDMA protocol based on message buffer reuse. Particularly, we use RDMA Direct protocols for processes with higher buffer reuse, and

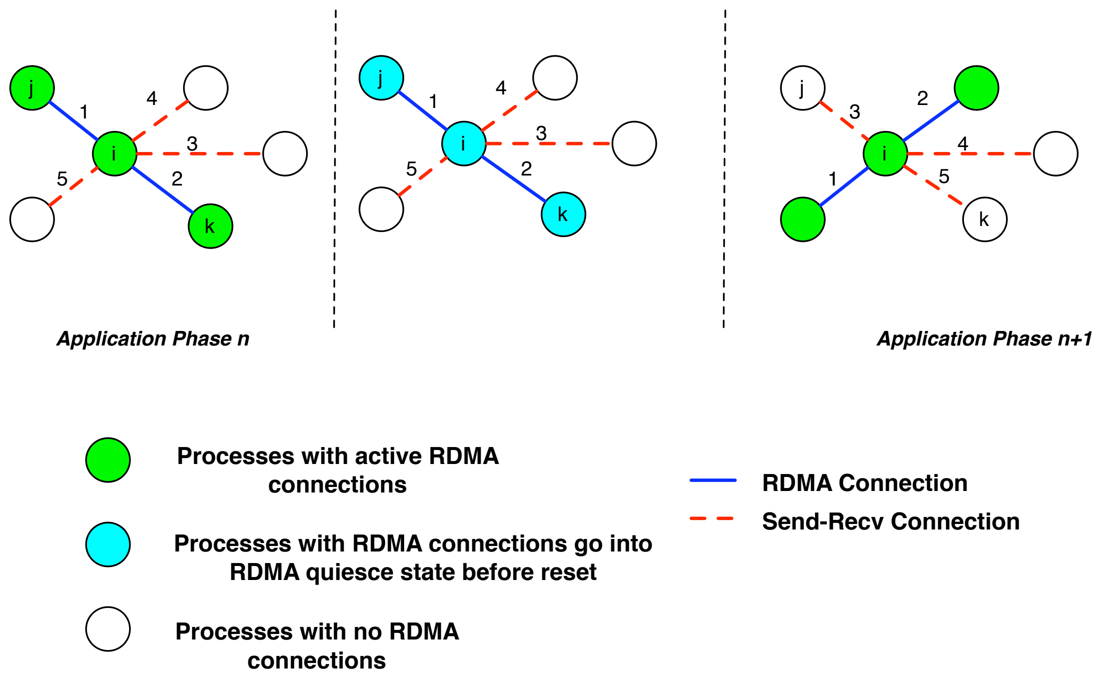

j
i
k
1
2
3
4
5
Application Phase n
Application Phase n+1
Processes with active RDMA connections
Processes with RDMA connections go into RDMA quiesce state before reset
Processes with no RDMA connections
RDMA Connection
Send-Recv Connection

Figure 5.3: Example connection reconfiguration between RDMA and send/receive connections. The nodes in the figure represent MPI processes, while the number on the edges represent the communication priority of the processes during an application phase.

RDMA Pipeline protocol for processes with lower buffer reuse. In this reconfiguration, in a profile run, the profiler gathers communication activity between processes during each application phase.

In subsequent runs on related inputs, the reconfiguration manager controls which RDMA large message protocol is used to communicate with each peer in each application phase using a schedule generated from the profiling information. This allows PRO-MPI to dynamically control which peers use RDMA Direct, improving performance and limiting registered buffer usage.

The profiling information and analysis needed for this reconfiguration is similar to the one used in connection-protocol reconfiguration. The profiler gathers the number of messages sent to each peer by each process during each application phase. The analyzer uses the profile information to generate a *priority matrix* as well as the maximum number of processes (*maxProc*) that can use RDMA Direct protocol. The three-dimensional priority matrix contains the communication priority between any two MPI processes during each application phase; specifically element $p[i][j][k]$ of the priority matrix contains the priority of communication from process i to j at application phase k.

The reconfiguration manager controls the process to use RDMA Direct based on the priority of process pair and available registered memory. At each reconfiguration point, two processes p and q, may use RDMA Direct, if the priority of that connection is less than *maxProc* and registered buffers used is less than *maxThreshold*. As a result, the available registered memory is allocated to only processes that have higher communication activity and higher probability of buffer reuse.

5.5 Evaluation

To demonstrate the capabilities of Open PRO-MPI we ran experiments to measure the impact of profile-based protocol reconfigurations on both synthetic microbenchmarks and application performance. In this section, we describe the experimental testbed used for the experiments, briefly describe the benchmarks and application, and present experimental results and provide analysis.

5.5.1 Setup

We ran all our experiments on the Pequena cluster at University of New Mexico Center for Advanced Research Computing. The cluster is a 22 node Altix ICE system with an Infiniband interconnect [73]. Each compute node has two 3.0 GHz

Intel Xeon processors each with four cores, 16GB of memory, and runs SUSE Linux with kernel version 2.6.16 and OpenFabric's OpenIB network stack. We compared the PRO-MPI MPI implementations with Open MPI (version 1.4).

5.5.2 Benchmark and Applications

- **Micro-benchmark:** To understand and estimate the performance benefits of using connection-protocol reconfiguration, we implemented a synthetic benchmark that imitates changing communication characteristics of an application, and then compared its performance while using profiled PRO-MPI and Open MPI.

  The benchmark can change communication characteristics such as message size and distribution, processes participating in a communication, and messages exchanged between a process pair. It can also define application steps, and can vary communication characteristics between the time steps.

- **HPC Challenge:** This is an application benchmark for measuring various aspects of machine (clusters, supercomputers) performance like FLOP/s, memory bandwidth, memory read/write, network latency, and bandwidth [76]. It consists of 7 tests : PTRANS, DGEMM, STREAM, HPL, RandomAccess, FFT, Communication bandwidth and latency.

  In our experiments, we use PTRANS and MPI_FFT (a parallel version of FFT) to measure the performance impact of buffer-registered reconfiguration. PTRANS is a test that measures memory bandwidth of the system by transferring a large array of data from multiple processor's memory. MPI_FFT is a test that measures floating point rate of execution by computing the fast Fourier transform operation.

- **SAMRAI** is a popular C++ software framework developed to implement parallel adaptive multi-physics applications [77]. It has been used to simulate black holes, dispersion of airborne materials, spread of fire, and interaction of laser and liquid plasma.

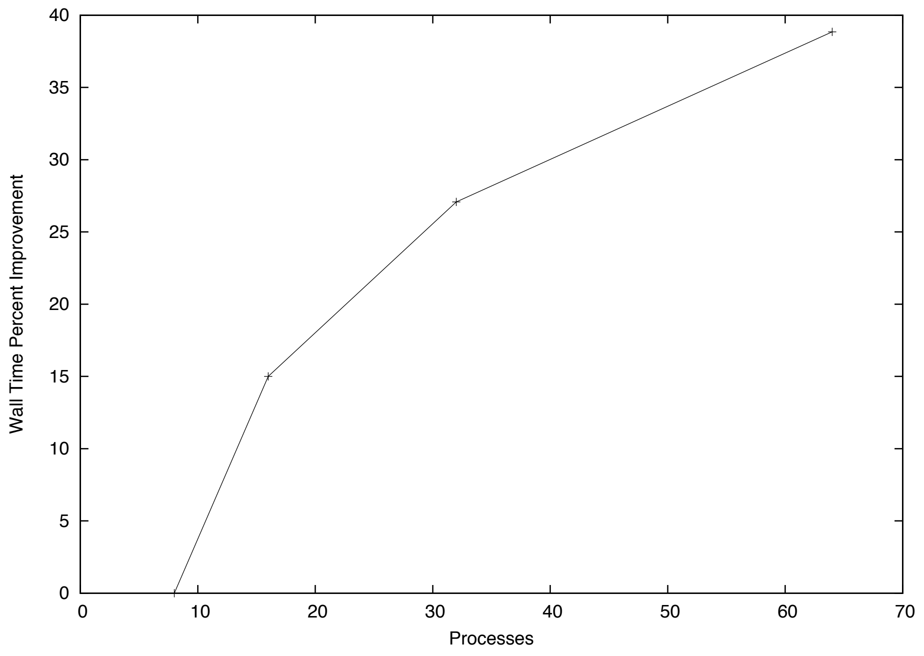
In our experiments, we ran SAMRAI's spherical shock wave problem for 10 time steps with the coarsest domain fixed to $[40, 40, 40]$ and varying refinement levels. This problem has dynamic communication characteristics; in particular, the processes that communicate change in each time step [10].

- **HyperCLaw** is a hybrid C++/Fortran AMR code developed and maintained by Lawrence Berkeley National Laboratory [78]. Our experiments used an AMR gas dynamics application data set which models the interaction of a Mach 1.25 shock in air hitting a helium bubble for 10 time steps with base grid of size $[32, 8, 8]$.

5.5.3 Synthetic Benchmark Results

To understand the performance impact on applications with changing communication characteristics when using reconfigurable libraries, we measured synthetic benchmark performance with Open PRO-MPI and Open MPI. Figure 5.4 and 5.5 shows that protocol reconfiguration increases in effectiveness with both increasing number of processes and messages sent per process, with 40% performance improvements over OpenMPI's static choice of connections to send-receive or RDMA protocol.

Figure 5.4, in particular, shows the impact of connection-protocol reconfiguration on the benchmark's runtime as the number of processes participating in the communication increases. Each data point in the Figure 5.4 shows the performance improvement of PRO-MPI over Open MPI for a certain problem size. For each data point, we ran the benchmark for 10 time steps, where the benchmark was configured to imitate ideal communication characteristics for PRO-MPI–messages exchanged

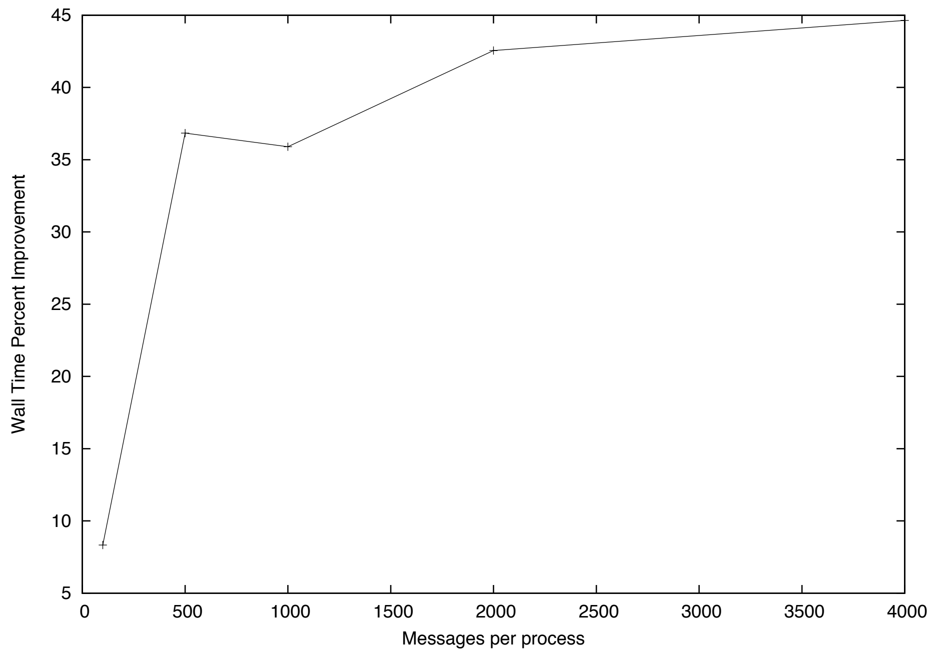


Increasing number of processes where each process sends 1000 msgs of size 1KB

Figure 5.4: PRO-MPI performance improvement over Open MPI on a synthetic communication benchmark

between any two process changed during time step 1 and other time steps. For these characteristics, Open MPI assigns RDMA connections to process pairs that exchange messages during startup. Although PRO-MPI assignment is similar to Open MPI for the first time step, it changes the assignment for other time steps. It assigns RDMA connections based on the process priorities (messages exchanged). As a result, PRO-MPI utilizes RDMA resources more efficiently.

Figure 5.5 shows the impact of reconfiguration when the number of messages sent by each process increases. For all data points in Figure 5.5, the problem size was



Increasing messages sent per process in a 32-process problem

Figure 5.5: PRO-MPI performance improvement over Open MPI on a synthetic communication benchmark

kept constant (32-process problem), and the benchmark was configured to imitate communication characteristics similar to the experiment in Figure 5.4. However between data points we increased the messages exchanged between a process pair. From Figure 5.5, we observe the protocol reconfiguration increases the performance with increasing number of messages per process.

5.5.4 Application Performance with Connection-protocol Reconfiguration and Exact Profiles

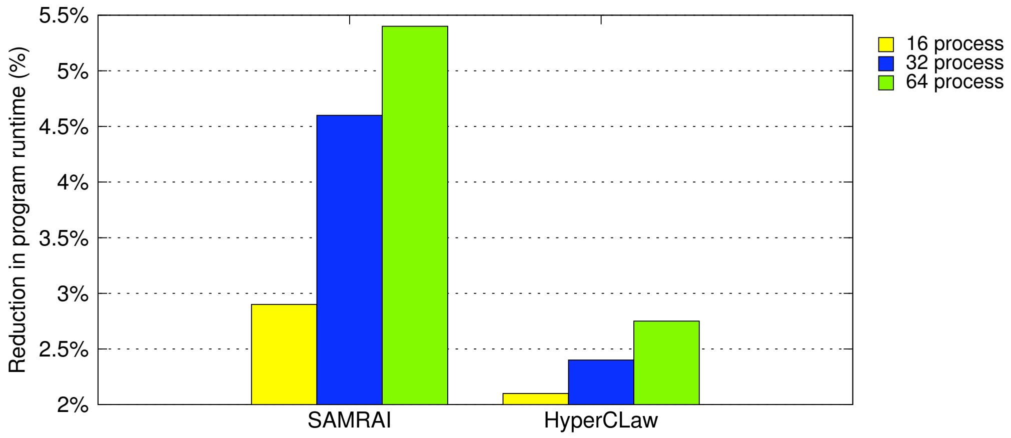


Figure 5.6: SAMRAI and HyperCLaw performance improvement using PRO-MPI (connection-protocol reconfiguration) with exact profiles

To measure connection-protocol reconfiguration impact on the performance of SAMRAI and HyperCLaw applications at different node counts, we compared the performance of Open MPI runs and profiled PRO-MPI runs of these applications at low refinement levels on 16, 32, and 64 nodes. Low mesh refinement levels were used both to limit application runtime. Each test was run 5 times.

Figure 5.6 shows the performance impact of connection-protocol reconfiguration on SAMRAI and HyperCLaw problems with exact performance profiles. SAMRAI performance is improved by 5.5% for a 64 process problem, and the performance improvement increases as the number of processes in the problem increases. Similarly, HyperCLaw performance improves by 2.7% for the 64 process problem, and the performance improvement increases slowly with increasing process counts.

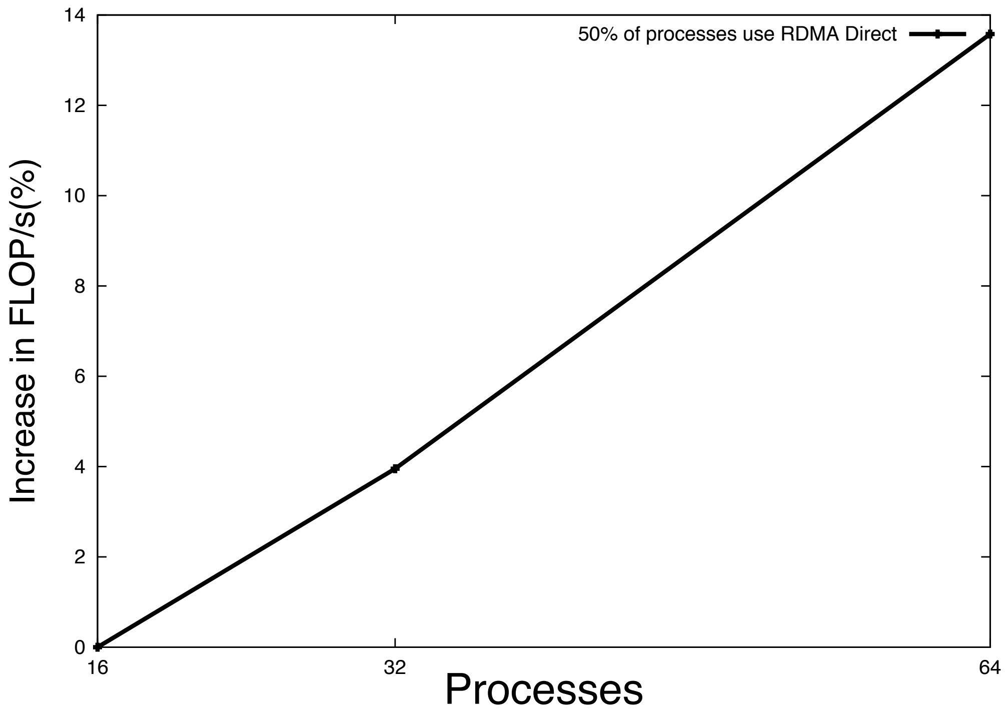


Figure 5.7: FFT performance improvement using PRO-MPI (registered-buffer reconfiguration) with exact profiles

Analysis

To understand the source of the observed performance improvements we profiled the communication characteristics of the applications while they used PRO-MPI and Open MPI for communication. As shown in Figure 5.8, on 64 nodes, while using exact profiles, HyperCLaw sends up to 20% RDMA messages when using PRO-MPI and up to 1% when using Open MPI with the same number of allowed RDMA connections. Similarly, SAMRAI sends 13 times more RDMA messages when using PRO-MPI instead of Open MPI with the same number of RDMA connections. This demonstrates that PRO-MPI's performance improvements are the result of more efficient use of RDMA connections compared to random assignment of RDMA connections to peers.

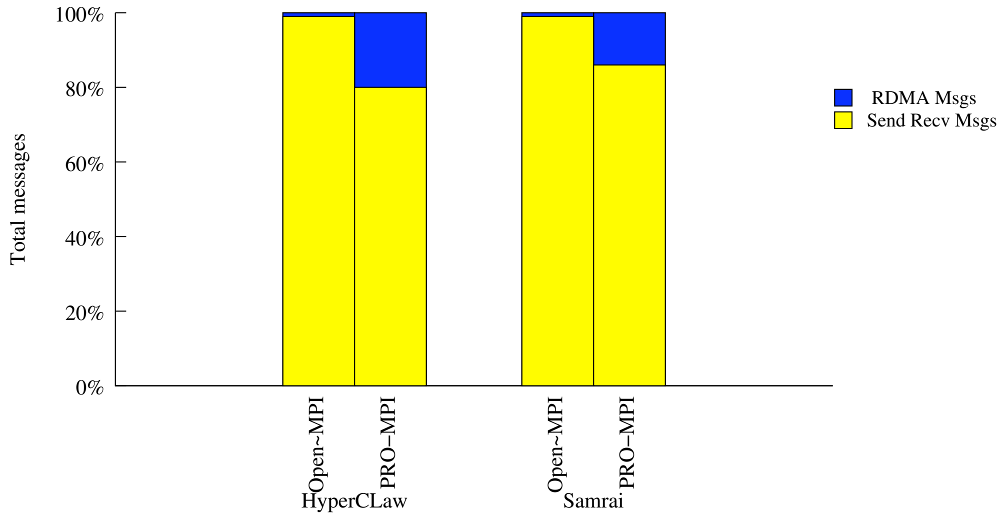


Figure 5.8: PRO-MPI communication chracteristics of HyperCLaw and SAMRAI while using PRO-MPI and Open MPI.

Although using PRO-MPI for communication has performance benefits, it adds overhead of creating and destroying RDMA connections at every reconfiguration point. Also, using RDMA requires dedicated memory buffers and a certain amount of polling overhead. Our results show that these overheads can be amortized when sufficiently many messages are exchanged between processes during each application phase. Also, applications can realize significant performance benefits while still using a relatively low $maxRDMA$ value in order to limit both memory usage and polling overhead. For our tests, we used $maxRDMA = 4$; attempting to use more RDMA connections (e.g. $maxRDMA = 8$) did not improve application performance in tests we ran.

5.5.5 Application Performance with Registered-buffer Reconfiguration and Exact Profiles

To measure the impact of registered-buffer reconfiguration on application performance, we compared the performance of the HPC Challenge benchmark on Open MPI and Open PRO-MPI. Because of its higher memory use, we used the benchmark's MPI_FFT kernel to measure the performance impact [79].

Figure 5.7 shows the performance impact of registered-buffer reconfiguration on MPI_FFT with exact performance profiles. For all data points, we limited caching of memory registrations to 250 MB. Therefore until the limit, the processes could use RDMA Direct protocol and after this limit, the processes send all messages using RDMA Pipeline protocol. Since there was limited registered memory, processes were prioritized for using RDMA Direct protocol. The priority was based on messages exchanged between process pairs. For all data points in the figure 5.7 *maxProc* was fixed to half the problem size (for example, 16 for a 32-process problem). This would maximize the use of registered buffers more efficiently. In the figure, we could observe that MPI_FFT performance was improved by 14% when 50% of processes could use RDMA Direct protocol and rest of the processes were using only RDMA Pipeline protocols.

5.5.6 Application Performance with Inexact Profiles

In Sections 5.5.4 and 5.5.5, we showed the performance impact of protocol reconfigurations using exact profiles. However, exact profiles may not be available or may be difficult or time-consuming to produce. We show experiments studying how profiles from one run can be used to improve the performance of an application on related input data.

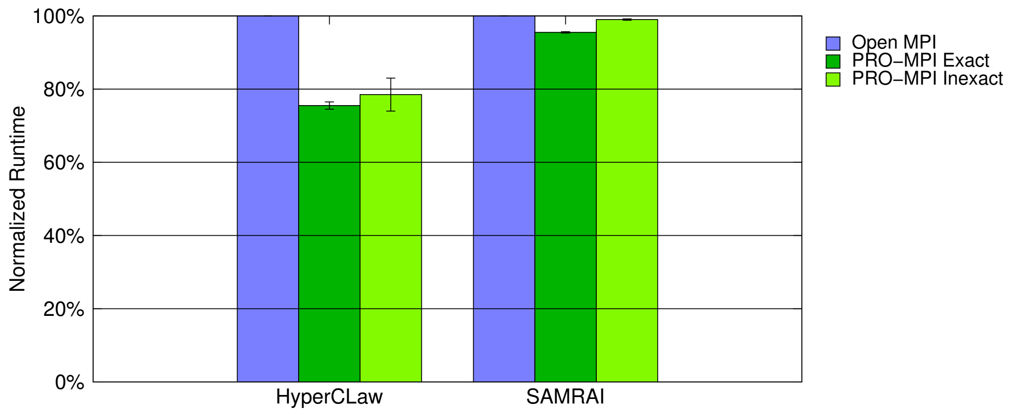


Figure 5.9: Performance improvement using profile collected at low mesh refinement (3 for SAMRAI, 2 for HyperCLaw) to optimize communication in application run with higher mesh refinement (4 for SAMRAI, 3 for HyperCLaw) on 32 nodes. Numbers represent the average of 5 runs.

Figure 5.9 shows that performance profiles collected from lower-refinement runs can be used to substantially improve the performance of more time-consuming high-refinement runs. Specifically, profiles from refinement 2 HyperCLaw runs can be used to improve the performance of refinement 3 HyperCLaw runs by approximately 16%. We also observed performance improvement in SAMRAI runs, though more

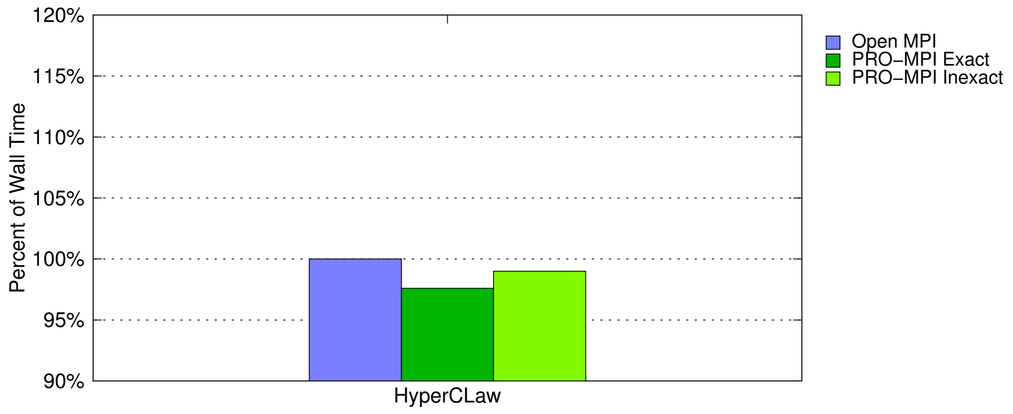


Figure 5.10: Performance improvement using profile collected at low problem size (processes=16) to optimize communication in application run with higher problem size(processes=32). Numbers represent the average of 5 runs.

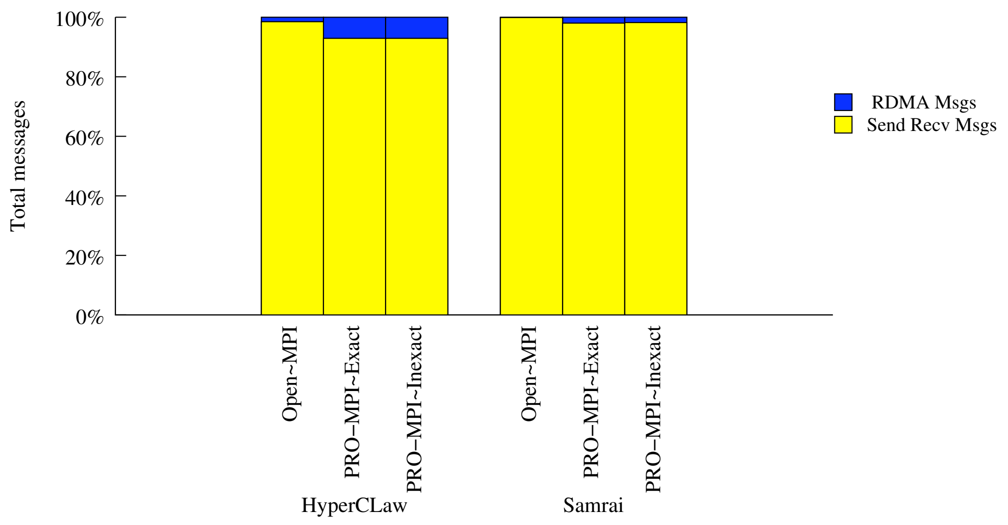


Figure 5.11: Communication chracteristics of HyperCLaw and SAMRAI while using Exact and Inexact profiles. Profiles of lower refinement level where used to drive a reconfiguration in higher refinement level problem.

modest ones than those observed in HyperCLaw.

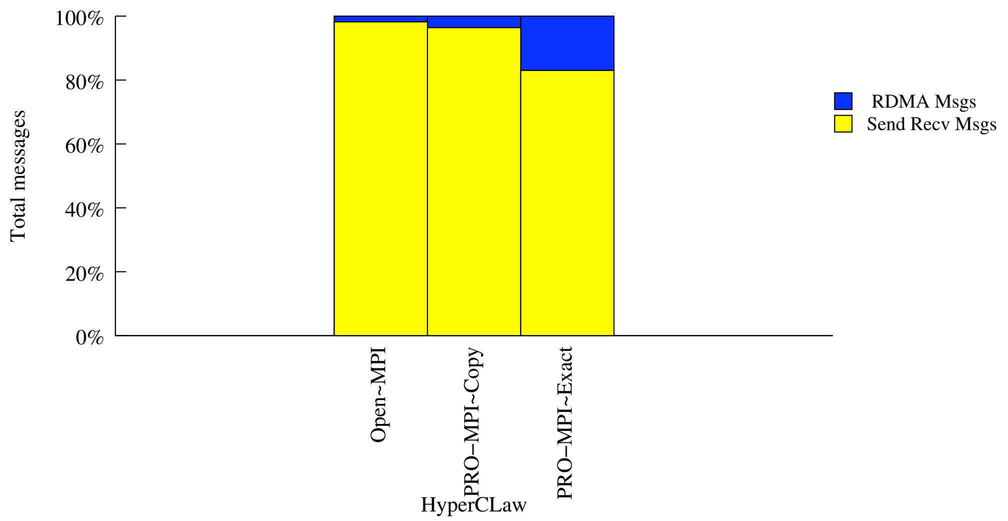


Figure 5.12: Communication chracteristics of HyperCLaw while using Exact and Inexact profiles. Profiles of lower problem size (procs =16) where used to drive a reconfiguration in higher problem size (procs =32).

Figure 5.10 shows the performance improvement, when profiles collected from smaller problem size is used to drive the reconfiguration of a larger problem size. Particularly, profiles from a 16-process HyperCLaw were used to improve the performance of 32-process HyperCLaw. To generate a profile for a 32-process problem from 16-process problem, the profiler collected profiles of a n-process HyperCLaw problem, and then the offline analyzer converted the profile to a 32-process problem size profile by extrapolating the missing data.

Analysis

To understand the performance improvement, as in Sections 5.5.4 and 5.5.5, we measured the percentage of RDMA and send/recv messages for each experiment. Figure 5.11 compares communication characteristics of HyperCLaw and SAMRAI while using no reconfiguration, and while using protocol reconfiguration with exact and inexact profiles. Figure 5.12 compares communication characteristics of HyperCLaw for a 32-process problem while using no reconfiguration, and using protocol reconfiguration with exact and inexact profiles

As expected, the performance improvement was the result of using RDMA connections efficiently. Comparing HyperCLaw results in Figure 5.9 and its characteristics in Figure 5.11, we can observe that performance improvement and communication characteristics are similar while using exact and inexact profiles. Both while using exact and inexact profiles, the percentage of RDMA messages was similar, and was proportional to the performance improvement. Similarly, the performance improvement for HyperCLaw while using inexact profiles derived from smaller problem size, was proportional to percentage of RDMA messages. However, for SAMRAI the percentage difference of RDMA messages was very low (approximately 0.2 %), and the performance difference was rather high (4%). This we believe is a consequence of SAMRAI being latency sensitive.

5.5.7 Reconfiguration Overhead

Both connection-protocol and register-buffer reconfigurations add overhead to the application runtime. This section discusses the overhead added by each reconfiguration, and analyzes when enabling reconfigurations can lead to the improvement in application performance.

Connection-protocol Reconfiguration Overhead

In the connection-protocol reconfiguration, the overhead is the result of establishing and reseting the RDMA connections at each reconfiguration point. In Open PRO-MPI, the RDMA connections are established by allocating the receive fragments and registered buffers at an origin process, and then sending this information to a destination process. To reset a RDMA connection, the origin process frees the receive fragments and registered buffers. It needs to acquire a RDMA lock to free the registered buffers and the receive fragments; the RDMA lock provides exclusive access to the buffers and also ensures that there are no waiting receives on the buffer. Besides the RDMA lock, it requires the endpoint lock to change the state of the process, and the Infiniband lock to update the number of RDMA connections.

When the connection-protocol reconfiguration is enabled in Open PRO-MPI, the reconfiguration overhead for the origin process to establish n RDMA connections at m reconfiguration points is:

$$RDMA^{m}_{connect} = m * n * L_c; Latency\ of\ a\ control\ message$$

And, the overhead to reset n RDMA connections is:

$$W_{Lock} = W_{RDMA} + W_{endpoint} + W_{Infiniband}$$

Where:

$W_{RDMA}, W_{endpoint}, W_{Infiniband}$ - the wait times for acquiring the RDMA lock, the endpoint lock and the Infiniband lock.

Besides the above overhead, during reconfiguration, all messages that were to be sent on RDMA connections are sent using send/receive protocol. As a result, this increases the latency of the messages. Also, some of the messages are delayed as the locks are acquired by the reconfiguration manager for reconfiguration. This also increases the latency of the messages. However, since Open PRO-MPI implementations are configured to reconfigure at the end of an application phase (where an application is assumed to reach a quiescent state), it is safe to assume that very few messages are exchanges during this time.

Overhead of connection-protocol reconfiguration when n RDMA connections are established, and reset at m reconfiguration points is:

$$Overhead_{connection-protocol} = R^{mn}_{connection} + W_{Lock}$$

As seen in this section, the connection-protocol reconfiguration improves the performance of an application by increasing the utilization of available limited RDMA connections. For the reconfiguration to improve the application performance, its should send more messages using the limited RDMA connections.

$$RDMA^{messages}_{connection-protocol} > RDMA^{messages}_{no-reconfiguration}$$

and

$$L^{'}_{RDMA} + L^{'}_{send/recv} - L_{RDMA} + L_{send/recv} > Overhead_{connection-protocol}$$

Where:

$L^{'}_{RDMA}, L^{'}_{send/recv}$ - the total latency of messages sent using RDMA and send/recv protocols while using the connection-protocol reconfiguration.

$L_{RDMA}, L_{send/recv}$ - the total latency of messages sent using RDMA and send/recv protocols with no reconfiguration enabled.

Registered-buffer Reconfiguration Overhead

In the registered-buffer reconfiguration, the reconfiguration overhead is the result of registering and de-registering memory buffers during a message transfer. At each reconfiguration point, all memory buffer registrations that are cached in the RCache are removed from the cache and de-registered. If any of these memory buffers are re-used by the application, they are re-registered during the message transfer and cached in the RCache. In Open MPI, however, when the RDMA Direct protocol is used these registered buffers would have not been removed from the cache or de-registered. Thus this reconfiguration adds the overhead which is de-registering and registering of a memory buffer.

$$Overhead_{registered-buffer} = n * t_s^{de-register} + \sum_{k=1}^{m} L_s^{k\ registering} - L_s^{k\ without\ registering}$$

This can be simplified to

$$Overhead_{registered-buffer} = n * t_s^{de-register} + m * t_s^{register}$$

Where:
n - the number of buffers that are de-registered at a reconfiguration point.
m - the number of buffers that are registered between any two reconfiguration points.
$t_s^{de-register}$, $t_s^{register}$ - the overhead for de-registering a buffer of size s.
$L_s^{k\ registering}$ - the latency of a message of size s along with the cost of registering its buffer.
$L_s^{k\ without\ registering}$ - the latency of a message of s without including the cost of registering its buffer.

Given the overhead of the registered-buffer reconfiguration, we can decide whether it is advantageous to use RDMA Direct protocol based on the buffer re-use characteristics.

$$L_s^{registering} - L_s^{without_registering} + t_{de-register} < R * (L_s^{RDMA\ Pipeline} - L_s^{RDMA\ Direct})$$

Where:

R - the number of times a memory buffer is used between any two reconfiguration points.

$L_s^{registering}$ - the latency of a message of size s along with the cost of registering its buffer.

$L_s^{without_registering}$ - the latency of a message of size s without including the cost of registering its buffer.

$L_s^{RDMA\ Pipeline}$ - the latency of a message of size s while using the RDMA Pipleline protocol.

$L_s^{RDMA\ Direct}$ - the latency of a message of size s while using the RDMA Direct protocol.

5.6 Summary

This chapter presented Open PRO-MPI, a implementation of PRO-MPI that supported full MPI-2 semantics. It also described two reconfigurations – connection-protocol reconfiguration and registered-buffer reconfiguration – that were implemented in Open PRO-MPI. The goal of connection-protocol reconfiguration was to improve application performance by increasing utilization of available limited RDMA connections, and the goal of registered-buffer reconfiguration was to improve application performance by taking advantage of limited registered buffers. The chapter also presented methods to generate application profiles for resource-intensive problems from the profiles of low-cost problems.

The chapter presented results showing the performance impact of using Open PRO-MPI and reconfigurations on HPC applications and benchmarks. The results demonstrated that reconfigurations improved HPC application and benchmark performance. It showed that exact-profile driven reconfiguration improved HyperCLaw performance by 22% and SAMRAI performance by 5.5%. It also showed that inexact-profile driven reconfiguration improved performance of HyperCLaw that was comparable to performance improvement obtained by exact-profile driven reconfiguration. However, inexact-profile driven reconfiguration improved SAMRAI performance only marginally.

Chapter 6

Conclusions

6.1 Summary

This dissertation presented PRO-MPI, a framework for constructing reconfigurable MPI implementations. Also, it demonstrated how reconfigurable MPI could be used to improve the performance of HPC applications and benchmarks with dynamic communication characteristics. To improve performance, PRO-MPI framework collects the applications characteristics to understand the application behavior, and drive a reconfiguration in MPI implementations. The goal of the reconfiguration is to tailor the protocol functionality of an MPI implementation to match application requirements.

In Chapter 2, we described prior work that dealt with constructing configurable protocol stacks, and support for reconfiguration and adaptation in current MPI implementations. None of these frameworks were geared towards constructing reconfigurable communication systems for high-performance computing. They all lacked support for high-performance network interfaces, and performance driven abstractions required. They were designed, developed, and implemented either for distributed network systems or mobile systems. All current MPI implementations, mentioned here, provided only a limited support for reconfiguration and adaptation; none of them enabled full-fledged support for functional reconfiguration.

Chapter 3 described the architecture of PRO-MPI framework. It comprised of four main components: profiler, analyzer, communication layer and reconfiguration manager. Each of these components had a specific functionality to realize profile-driven reconfigurable MPI implementation. The profiler collects application information to construct application communication profile, and the analyzer uses this profile to provide a series of changes to MPI protocol functionality. The communication layer provides an MPI implementation that is functionally modular. And, the reconfiguration manager ensures that protocol changes were executed on the communication layer.

Chapter 4 described Cactus PRO-MPI, which is a reconfigurable MPI implementation based on PRO-MPI framework that used CTP as a communication layer. CTP, a configurable transport protocol, based on Cactus system, allows protocol functionality to built from composable smaller modules. The functional composition provides the ability to mix and match the protocol functionality of the transport protocol, and also potentially the ability to change its functionality at runtime. The extensions and modifications, described in Chapter 4 to CTP enabled it to provide an reconfigurable MPI implementation. Additional extensions – profiler, analyzer and reconfiguration manager – enabled Cactus PRO-MPI to change its functionality in response to changing application characteristics. Also in this chapter, we presented results that demonstrated Cactus PRO-MPI could improve performance characteristics of HPC applications.

In Chapter 5, we described another PRO-MPI based reconfigurable MPI implementation, Open PRO-MPI. It used Open MPI as an communication layer implementing MPI. Open PRO-MPI supported full MPI-2 specification unlike Cactus

PRO-MPI, which only supported point-to-point MPI semantics. Addition of profiler, analyzer, and reconfiguration manager enabled profile-driven reconfiguration in Open PRO-MPI. Also, adding two reconfiguration implementations (connection-protocol reconfiguration and registered-buffer reconfiguration) demonstrated that this framework could be used to implement many reconfigurations. Using these reconfigurations, we demonstrated that Open PRO-MPI can improve the performance of HPC applications and benchmarks.

Finally, also in Chapter 5, we provided methods to construct the inexact application profiles from an exact application profile of the same application. Results in the Chapter 5 showed that these approximate profiles could be used to drive a reconfiguration and improve application performance.

6.2 Future Work

The work in the dissertation could be extended in number of different directions. Many new reconfigurations can be added to PRO-MPI that can improve the performance of other HPC applications. The framework can be extended to help construct reconfigurable communication libraries for multi-core and hybrid architectures. This section elaborates more on these directions.

By adding new reconfigurations, the reconfigurable MPI implementations could improve performance of many more HPC applications. Research has shown that performance of the applications can be improved by selecting algorithms for collective operations based on the machine topology and application characteristics such as message size and type [14, 16, 41]. This framework enables adding such reconfigurations to the current implementations, and studying the performance impact on the application performance.

With growing usage of multi-core processors in HPC machines, designing communication libraries that take advantage of these cores for parallelism is very important. PRO-MPI and the reconfigurable MPI implementations described in this dissertation may provide a better paradigm for constructing communication libraries for machines with multi-cores and hybrid architecture nodes. Research has demonstrated that event-based software design provides the concurrency and granularity to build efficient communication libraries for multi-core architectures [80, 81]. In Cactus PRO-MPI, which is an event-based MPI implementation, the smallest unit of functionality is implemented as a event handler. These handlers each can be mapped onto different POSIX-type threads. The threads that are independent can be mapped onto different processor cores for load balancing, improving parallelism or for the better utilization of computing power.

References

[1] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[2] Message Passing Interface Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.

[3] The MPI Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Application*, 8(3/4):165–416, 1994.

[4] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. Open MPI: A flexible high performance MPI. In *Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics*, Poznan, Poland, September 2005.

[5] Jeffrey M. Squyres and Andrew Lumsdaine. The component architecture of Open MPI: Enabling third-party collective algorithms. In Vladimir Getov and Thilo Kielmann, editors, *Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, pages 167–185, St. Malo, France, July 2004. Springer.

[6] William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[7] Fabrizio Gagliardi. HPC Opportunities and Challenges in e-Science. In *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part I*, pages 18–19, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] Earl Joseph, Jei Wu, Steve Conway, and Suzy Tichenor. Council on competitiveness study of innovation, competitiveness, and HPC, 2008.

[9] *http://www.top500.org/.*

[10] Jeffrey S. Vetter and Frank Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *J. Parallel Distrib. Comput.*, 63(9):853–865, 2003.

[11] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural requirements of parallel scientific applications with explicit communication. In *ISCA '93: Proceedings of the 20th annual international symposium on computer architecture*, pages 2–13, New York, NY, USA, 1993. ACM.

[12] Shoaib Kamil, Ali Pinar, Daniel Gunter, Michael Lijewski, Leonid Oliker, and John Shalf. Reconfigurable hybrid interconnection for static and dynamic scientific applications. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 183–194, New York, NY, USA, 2007. ACM.

[13] Manjunath Gorentla Venkata, Patrick G. Bridges, and Patrick M. Widener. Using Application Communication Characteristics to Drive Dynamic MPI Reconguration. In *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS),CAC Workshop*, Rome, Italy, May 2009.

[14] J Pjesivac-Grbovic, G Fagg, T Angskun, G Bosilca, and J Dongarra. MPI Collective Algorithm Selection and Quadtree Encoding. *Lecture Notes in Computer Science*, Vol. 4192, Number 2006:pp. 40–48, 2006.

[15] Manjunath Gorentla Venkata and Patrick G. Bridges. MPI/CTP: A reconfigurable MPI for HPC applications. In Dieter Kranzlmüller, Peter Kacsuk, Jack Dongarra, and Jens Volkert, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 13th European PVM/MPI Users' Group Meeting*, volume 4192 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

[16] George Almási, Philip Heidelberger, Charles J. Archer, Xavier Martorell, C. Chris Erway, José E. Moreira, B. Steinmacher-Burow, and Yili Zheng. Optimization of MPI collective communication on BlueGene/L systems. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 253–262, New York, NY, USA, 2005. ACM.

[17] David G. Solt. A profile-based approach for topology aware MPI rank placement. In *High Performance Computation Conference (HPCC), Springer Lecture Notes in Computer Sciences*, 2007.

[18] Norm Hutchinson and Larry L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[19] Norm Hutchinson, Larry L. Peterson, Sean O'Malley, and M. Abbott. RPC in the *x*-kernel: Evaluating new design techniques. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 91–101, Litchfield Park, AZ, 1989.

[20] Norm Hutchinson and Larry L. Peterson. The design of the *x*-kernel. In *Proceedings of ACM SIGCOMM '88*, pages 65–75, 1988.

[21] Matti A. Hiltunen and Richard D. Schlichting. The Cactus Approach to Building Configurable Middleware Services. In *Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, Nurnberg, Germany, 2000.

[22] Patrick G. Bridges, Matti A. Hiltunen, Richard D. Schlichting, Gary T. Wong, and Matthew Barrick. A configurable and extensible transport protocol. *ACM/IEEE Transactions on Networking*, 15(6):1254–1265, December 2007.

[23] R. X. Wu, A. A. Chien, M. A. Hiltunen, R. D. Schlichting, and S. Sen. A high performance configurable transport protocol for grid computing. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2*, pages 1117–1125, Washington, DC, USA, 2005. IEEE Computer Society.

[24] Hugo Miranda, Alexandre Pinto, and Luis Rodrigues. Appia: A Flexible Protocol Kernel Supporting Multiple Coordinated Channels. *Distributed Computing Systems, International Conference on*, 0:0707, 2001.

[25] Sergio Mena, Xavier Cuvellier, Christophe Grgoire, and Andr Schiper. Appia vs. Cactus: Comparing Protocol Composition Frameworks. In *Proceedings of 22th IEEE Symposium on Reliable Distributed Systems (SRDS03*, 2003.

[26] Hugo Miranda and Lus Rodrigues. Communication support for multiple QoS requirements, 1999.

[27] S. Michiels, F. Matthijs, D. Walravens, and P. Verbaeten. Position summary: DiPS: A unifying approach for developing system software. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, 2001.

[28] Sam Michiels and Petrus Verbaeten. Component Framework Technology for Flexible Protocol Stacks. In *Proceedings of the 1st Flanders Engineering PhD Symposium*, December 2003.

[29] Kenneth P. Birman, R. Renesse, and W. Vogels. The Ensemble distributed communication system. http://simon.cs.cornell.edu/Info/Projects/Ensemble/, 1996.

[30] Mark Hayden. The Ensemble system. Technical Report TR98-1662, Department of Computer Science, Cornell University, 1998.

[31] Robert van Renesse, Kenneth P. Birman, Mark Hayden, Alexey Vaysburd, and David A. Karr. Building adaptive systems using Ensemble. Technical report, Cornell University, Ithaca,NY, 1997.

[32] Galen Mark Shipman, Tim S. Woodall, George Bosilca andRich L. Graham, and Arthur B. Maccabe. High performance RDMA protocols in HPC. In *Proceedings, 13th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, Bonn, Germany, September 2006. Springer-Verlag.

[33] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[34] Galen M. Shipman, Timothy S. Woodall, Richard L. Graham, Arthur B. Maccabe, and Patrick G. Bridges. Infiniband scalability in OpenMPI. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[35] William D. Gropp and Ewing Lusk. *Installation Guide for* `mpich`*, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/5.

[36] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[37] Jiesheng Wu, Jiuxing Liu, P. Wyckoff, and D. Panda. Impact of on-demand connection management in MPI over VIA. *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 152–159, 2002.

[38] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[39] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on LanguaDynges and Compilers for Parallel Computing (LCPC 03)*, College Station, Texas, October 2003.

[40] M. Bhandarkar, L.V. Kale, E. de Sturler, and Jay Hoeflinger. Adaptive load balancing for MPI programs. In *Proceedings of International Conference on Computational Science*, pages 108–117, San Francisco, CA, 2001.

[41] Ahmad Faraj, Xin Yuan, and David Lowenthal. STAR-MPI: self tuned adaptive routines for MPI collective operations. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 199–208, New York, NY, USA, 2006. ACM.

[42] Ahmad Faraj and Xin Yuan. Automatic generation and tuning of MPI collective communication routines. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 393–402, New York, NY, USA, 2005. ACM.

[43] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffery K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[44] Kevin A. Huck, Allen D. Malony, Sameer Shende, and Alan Morris. TAUg: Runtime global performance data access using MPI. In *Proceedings of EuroPVMPI*, 2006.

[45] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP-19)*, Bolton Landing, NY, 2003.

[46] J. M. Anderson, L. M. Berc, S. Ghemawat J. Dean, M.R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Transactions on Computer systems*, 15(4):357–390, 1997.

[47] Jeffrey S. Vetter. Dynamic statistical profiling of communication activity in distributed applications. In *Proceedings of SIGMETRICS 2002*, 2002.

[48] Matthew J. Sottile and Ronald G. Minnich. Supermon: A high-speed cluster monitoring system. In *IEEE Conference on Cluster Computing*, September 2002.

[49] Matthew L. Massie, Brent N. Chun, and David E. Culler. The Ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, May 2004.

[50] Wenbin Zhu, Patrick G. Bridges, and Arthur B. Maccabe. Lightweight application monitoring and tuning with embedded gossip. *IEEE Transactions of Parallel and Distributed Systems (TPDS)*, 2008. Accepted for Publication.

[51] Wenbin Zhu, Patrick G. Bridges, and Arthur B. Maccabe. Light-weight application monitoring and tuning with embedded gossip. In *Proceedings of the 2007 SIGMETRICS Student Workshop*, June 2007.

[52] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: online modelling and performance-aware systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HOTOS-IX)*, Lihei, Hawaii, 2003.

[53] Clint Whaley Antoine, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27:2001, 2000.

[54] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE Intl. Conference on Acoustics, Speec, and Signal Processing (ICASSP '98)*, volume 3, pages 1381–1384, Seattle, WA, 1998.

[55] Jeff Bilmes, Krste Asanovic, Jim Demmel, Dominic Lam, and Chee-Whye Chin. Optimizing matrix multiply using PHiPAC: a Portable, High-performance, ANSI C Coding methodology. Technical report, 1996.

[56] Gregory R. Andrews, Saumya K. Debray, Benjamin W. Schwarz, and Matthew P. Legendre. Using link-time optimization to improve the performance of MPI Programs. Technical report, 2001.

[57] Wenbin Zhu, Patrick G. Bridges, and Arthur B. Maccabe. Online critical path profiling for parallel applications. In *Proceedings of the 2005 IEEE International Conference on Cluster Computing (Cluster 2005)*, Boston, MA, September 2005.

[58] David Kincaid and Ward Cheney. *Numerical Analysis: Mathematics of Scientific Computing.* 2002.

[59] Matti A. Hiltunen, Richard D. Schlichting, Xiaonan Han, Melvin Cardozo, and Rajsekhar Das. Real-time dependable channels: Customizing QoS attributes for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):600–612, 1999.

[60] Matti A. Hiltunen, Gary T. Wong, and Richard D. Schlichting. Dynamic messages: An abstraction for complex communication protocols. *Software: Practice and Experience*, 2001. Submitted for publication.

[61] Gary T. Wong, Matti A. Hiltunen, and Richard D. Schlichting. A configurable and extensible transport protocol. In *Proceedings of IEEE INFOCOM '01*, pages 319–328, Anchorage, Alaska, 2001.

[62] Myricom, Inc. *LANai 7*, 1999. http://www.myri.com/vlsi/LANai7.pdf.

[63] Myricom, Inc. *The GM API*, 1999.

[64] Myricom, Inc. *PCI64 Programmer's Documentation*, 2001. http://www.myri.com/myrinet/PCI64/programming.html.

[65] Myricom, Inc. The GM message passing system. Technical report, Myricom, Inc., 1997.

[66] *http://mvapich.cse.ohio-state.edu/benchmarks*.

[67] Ron Brightwell and Keith Underwood. Evaluation of an eager protocol optimization for MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: Tenth European PVM/MPI Users' Group Meeting*, / 2003.

[68] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[69] Los Alamos National Laboratories. The Los Alamos Message Passing Interface. http://public.lanl.gov/lampi/.

[70] Graham E. Fagg and Jack J. Dongarra. FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world, 2000.

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

[72] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, /2002.

[73] InfiniBand Trade Association. *http://www.infinibandta.org*, 1999.

[74] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High performance RDMA-Based MPI implementation over Infiniband. In *In 17th Annual ACM International Conference on Supercomputing (ICS 03*, 2003.

[75] Wenbin Zhu. OS bypass investigation and experimentation. Master's thesis, The University of New Mexico, Computer Science Department, Albuquerque, NM 87131, 2002.

[76] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC Challenge (HPCC) benchmark suite. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 213, New York, NY, USA, 2006. ACM.

[77] Andrew M. Wissink, Richard D. Hornung, Scott R. Kohn, Steve S. Smith, and Noah Elliott. Large scale parallel structured AMR calculations using the SAMRAI framework. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 6–6, New York, NY, USA, 2001. ACM.

[78] Charles A. Rendleman, Vincent E. Beckner, Mike Lijewski, William Crutchfield, and John B. Bell. Parallelization of structured, hierarchical adaptive mesh refinement algorithms, 1999.

[79] Shuaiwen Song, Rong Ge, Xizhou Feng, Cameron, and Kirk W. Energy Profiling and Analysis of the HPC Challenge Benchmarks. *Int. J. High Perform. Comput. Appl.*, 23(3):265–276, 2009.

[80] Supratik Majumder and Scott Rixner. An event-driven architecture for MPI Libraries. In *In Proceedings of the 2004 Los Alamos Computer Science Institute Symposium*, 2004.

[81] François Trahay, Élisabeth Brunet, Alexandre Denis, and Raymond Namyst. A multithreaded communication engine for multicore architectures. In *CAC 2008: Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2008*, Miami, FL, April 2008. IEEE Computer Society Press.