

**Implementation of MP\_Lite for the VI Architecture**

by

Weiyi Chen

A thesis submitted to the graduate faculty  
in partial fulfillment of the requirements for the degree of  
MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:  
Ricky A. Kendall, Co-major Professor  
Srinivas Aluru, Co-major Professor  
Lu Ruan

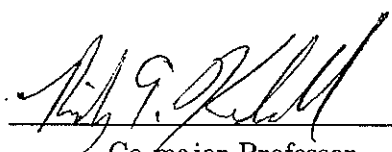
Iowa State University

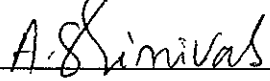
Ames, Iowa

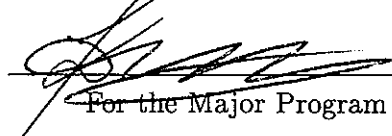
2001

Graduate College  
Iowa State University

This is to certify that the Master's thesis of  
  
Weiyi Chen  
  
has met the thesis requirements of Iowa State University

  
\_\_\_\_\_  
Co-major Professor

  
\_\_\_\_\_  
Co-major Professor

  
\_\_\_\_\_  
For the Major Program

## TABLE OF CONTENTS

<b>ACKNOWLEDGMENTS</b> . . . . .	viii
<b>ABSTRACT</b> . . . . .	ix
<b>CHAPTER 1. INTRODUCTION</b> . . . . .	1
1.1 Parallel Computing . . . . .	1
1.2 Communication in Parallel Computers . . . . .	2
1.3 The Goal of This Thesis . . . . .	3
1.4 Organization . . . . .	4
1.5 Other Research Efforts . . . . .	4
<b>CHAPTER 2. COMMUNICATION WITHIN A CLUSTER</b> . . . . .	7
2.1 Parallel Virtual Machine . . . . .	7
2.2 Message Passing Interface . . . . .	8
2.2.1 MPICH . . . . .	10
2.2.2 LAM MPI . . . . .	11
2.2.3 MP_Lite . . . . .	11
2.3 Other Message-Passing Libraries . . . . .	14
2.4 User-level Networking . . . . .	15
2.4.1 Active Messages . . . . .	15
2.4.2 U-Net . . . . .	16
2.4.3 Fast Messages . . . . .	16
2.4.4 Virtual Memory-Mapped Communication . . . . .	17
2.4.5 Basic Interface for Parallelism . . . . .	17
2.4.6 Scheduled Transfer Protocol . . . . .	17

2.4.7	Virtual Interface Architecture . . . . .	18
2.5	VIA Implementations . . . . .	22
2.5.1	M-VIA . . . . .	22
2.5.2	The Berkeley VIA Implementation . . . . .	23
2.5.3	Commercial Products . . . . .	23
2.6	VIA Implementations for MPI . . . . .	23
2.6.1	MVICH . . . . .	23
2.6.2	M-VIA for LAM MPI . . . . .	24
2.6.3	VIA for MPI/PRO . . . . .	25
2.6.4	MPI Implementation on the NTSC VIA cluster . . . . .	25
2.6.5	MP_Lite M-VIA . . . . .	25
<b>CHAPTER 3. IMPLEMENTATION OF MP_LITE FOR M-VIA . . . . .</b>		<b>26</b>
3.1	System Overview . . . . .	26
3.2	Queue Management . . . . .	28
3.3	Buffer Management . . . . .	29
3.4	Important Data Structures . . . . .	31
3.5	Initialization . . . . .	32
3.6	Communication Protocols . . . . .	34
3.6.1	The Eager Protocol . . . . .	35
3.6.2	The Handshake Protocol . . . . .	36
3.7	Dynamic Memory Registration . . . . .	38
3.8	Send . . . . .	38
3.9	Receive . . . . .	40
3.10	Channel-Bonding . . . . .	42
3.11	Finalization . . . . .	44
3.12	Porting M-VIA to the Alpha Platform . . . . .	44
<b>CHAPTER 4. PERFORMANCE OF MP_LITE M-VIA ON LINUX . . . . .</b>		<b>46</b>
4.1	Experimental Environment . . . . .	46

4.1.1	Configuration . . . . .	46
4.1.2	NetPIPE Performance Evaluator . . . . .	47
4.2	Point-to-Point Communication . . . . .	47
4.2.1	Fast Ethernet on the PC Cluster . . . . .	47
4.2.2	Gigabit Ethernet on the PC Cluster . . . . .	49
4.2.3	Gigabit Ethernet on the Alpha Cluster . . . . .	51
4.3	Channel-Bonding on Linux Clusters . . . . .	52
4.4	Summary . . . . .	53
<b>CHAPTER 5. DISCUSSION AND CONCLUSIONS . . . . .</b>		<b>55</b>
5.1	Features . . . . .	55
5.1.1	High Performance . . . . .	55
5.1.2	Channel-Bonding . . . . .	56
5.1.3	Portability . . . . .	56
5.1.4	User Friendly System . . . . .	57
5.2	Limitations . . . . .	57
5.2.1	Reliability . . . . .	57
5.2.2	Resource Reservation . . . . .	59
5.2.3	Channel-Bonding Issues . . . . .	60
5.2.4	Overlapping Communication and Computation . . . . .	60
5.2.5	Other Issues . . . . .	61
5.3	Conclusions and Future Efforts . . . . .	62
<b>BIBLIOGRAPHY . . . . .</b>		<b>65</b>

## LIST OF TABLES

Table 3.1	Memory copy compared to memory registration . . . . .	36
Table 4.1	Test cluster configuration . . . . .	46
Table 4.2	Installed M-VIA implementation for MPI . . . . .	46
Table 5.1	Reliability guarantees . . . . .	58

## LIST OF FIGURES

Figure 2.1	Diagram of the structure of MP_Lite . . . . .	13
Figure 2.2	VI architecture model . . . . .	19
Figure 2.3	A Virtual Interface . . . . .	20
Figure 3.1	MP_Lite M-VIA module overview . . . . .	27
Figure 3.2	An example of the receive queue . . . . .	29
Figure 3.3	mbufs . . . . .	30
Figure 3.4	Diagram of the eager protocol . . . . .	35
Figure 3.5	Diagram of the handshake protocol . . . . .	37
Figure 3.6	Channel-bonding for small messages . . . . .	43
Figure 3.7	Channel-bonding for large messages sent by the RDMA Write . . . . .	43
Figure 4.1	The throughput between Tulip Fast Ethernet cards on two PCs . . . . .	48
Figure 4.2	The communication latency between Fast Ethernet cards . . . . .	49
Figure 4.3	The throughput between Syskonnect Gigabit Ethernet cards on the PC test cluster . . . . .	50
Figure 4.4	The throughput as a function of message size on the Alpha cluster . . . . .	51
Figure 4.5	The signature graph on the Alpha cluster . . . . .	52
Figure 4.6	Channel-bonding up to four 3Com Fast Ethernet cards between PCs . . . . .	53
Figure 4.7	Channel-bonding two Gigabit Ethernet cards on the Alpha cluster . . . . .	54

## ACKNOWLEDGMENTS

I am grateful to many people who helped me in my research and the writing of this thesis. Thanks to my advisors Dr. Ricky A. Kendall and Dr. David E. Turner for their guidance, patience, mentoring, and support during my research work at the Scalable Computing Laboratory and the preparation of this thesis. My committee members Dr. Srinivas Aluru, Dr. Lu Ruan and Dr. Don E. Heller also gave me much help and advice.

Thanks to Mike Welcome and Paul Hargrove of NERSC for several helpful conversations about the M-VIA and MVICH projects. I would also like to give my special thanks to SCL secretary Vicki O'Neal for correcting grammar errors in this thesis.

This work was performed at Ames Laboratory under contract NO. W-7405-Eng-82 with the U.S. Department of Energy. The United States government has assigned the DOE report number IS-T 2030 to this thesis.



## ABSTRACT

MP\_Lite is a light weight message-passing library designed to deliver the maximum performance to applications in a portable and user friendly manner. The Virtual Interface (VI) architecture is a user-level communication protocol that bypasses the operating system to provide much better performance than traditional network architectures. By combining the high efficiency of MP\_Lite and high performance of the VI architecture, we are able to implement a high performance message-passing library that has much lower latency and better throughput.

The design and implementation of MP\_Lite for M-VIA, which is a modular implementation of the VI architecture on Linux, is discussed in this thesis. By using the eager protocol for sending short messages, MP\_Lite M-VIA has much lower latency on both Fast Ethernet and Gigabit Ethernet. The handshake protocol and RDMA mechanism provides double the throughput that MPICH can deliver for long messages. MP\_Lite M-VIA also has the ability to channel-bonding multiple network interface cards to increase the potential bandwidth between nodes. Using multiple Fast Ethernet cards can double or even triple the maximum throughput without increasing the cost of a PC cluster greatly.

## CHAPTER 1. INTRODUCTION

### 1.1 Parallel Computing

The need for more computational power is the main driving force in the development of computers. Scientific and engineering problems require extremely fast computers to simulate physical phenomena. Some typical examples include weather prediction, the atomic structure of materials, the evolution of galaxies and the behavior of microscopic electronic devices (CS99). To satisfy the computation need, one approach is to build a more powerful processor and use a huge amount of memory. However, a single processor in many cases still cannot meet the computational demand. For example, it will take 13 hours to predict the earth's weather for the next two days by using a computer that can execute one trillion ( $10^{12}$ ) calculations per second (Pac97). Moreover, the speed of light is an intrinsic limitation to the speed of computers (Dem95). Instead of using a more powerful single processor, another solution is parallel computing: use multiple-cooperating processors to solve large problems.

There are two broad classes of parallelism: SIMD (Single Instruction Multiple Data) and MIMD (Multiple Instruction Multiple Data). SIMD systems perform the same operation on different data concurrently. Vector machines, such as Cray T-90, and systems like the CM2 are examples of the SIMD architectures. MIMD systems perform different operations on different data concurrently. MIMD architectures have two basic types: shared-memory or distributed-memory.

Shared-memory MIMD architectures consist of a collection of processors and memory modules that share the same memory bus. Each processor can access any memory module directly. Although the memory access is faster than distributed memory computers, shared-memory systems have specific problems such as memory consistency. Examples of shared-memory

architectures include the SGI Origin systems, IBM 43p/44p and Compaq ES40/DS20.

In distributed-memory MIMD systems, each processor has its own private memory. Access to other processors and memory is via the network. There are many network interconnect topologies such as 2D and 3D meshes, fat trees, and flat networks. Examples of distributed-memory architectures include the Cray T3E, Intel Paragon and IBM SP-2.

Distributed-memory MIMD systems can also be built using a group of PCs or workstations. Such systems are referred to as clusters. A cluster is a collection of independent computer systems tightly-coupled by a dedicated network to form a multiprocessor computing environment. Building a cluster is very economical and can have significant computational power, but the network can limit the kind of applications that will run effectively on it.

## 1.2 Communication in Parallel Computers

Shared-memory computers typically use compiler directives that control concurrency and access to data or use a native shared-memory library for inter-process communication. Distributed-memory computers use a message-passing paradigm. Parallel computer vendors usually have their own message-passing libraries optimized for their particular machines. There are also many free distributions suitable for a variety of architectures. Among those implementations, MPICH (GLnDS96) and LAM MPI (BDV94) are the two most commonly used message-passing libraries that conform to the message passing interface (MPI) standard(For94).

The basic operations of message-passing are the *send* and *receive* functions. The simplest model to measure the communication cost for sending a message is: *communication time* = *latency* + *message size* / *bandwidth*. The communication time can make a big difference in the performance of a parallel application. Therefore, it is always desirable to improve the performance of the underlying message-passing library and the network protocol. The performance of a message-passing library or a network protocol is usually measured by three factors:

1. Latency: The preparation time for sending a message, or the time to send a smallest useful message. It can be roughly measured by sending and receiving a 1 byte message

to another node and dividing the round-trip time by 2. Latency has a significant impact on applications that pass small to moderately sized messages.

2. Bandwidth: The measurement of the communication rate. It tells us the maximum number of bits or bytes that can be transferred per second.
3. Host processing cost: The CPU cycles consumed for communication.

From a software perspective, the performance of a message-passing library can be improved in two ways: improve the performance of the message-passing layer or improve the underlying network protocol.

The implementation of traditional network protocols, such as TCP, suffers a performance penalty because of the operating system processing overhead and the extra memory-to-memory copies between kernel space and user space. Many research efforts have designed user-level protocols that bypass the operating system to deliver higher performance (vECgS92; vEBBV95; PKC97; DBL<sup>+</sup>97; PT98; fIT00; CCC97). The Virtual Interface Architecture (CCC97) is one such protocol that defines an interface between high performance network hardware and computer systems.

The message-passing layer can also be improved. For portability issues, many message-passing implementations complicate the internal queuing structure and require extra buffering for normal operations. Therefore, they do not match the performance of the underlying network protocol. MP\_Lite (Tur) is a light weight message-passing library designed to streamline the flow of data and deliver the maximum performance to the application.

### 1.3 The Goal of This Thesis

It is clear that if we can integrate the advantages of a light-weight message-passing library and OS-bypass network protocols, we can implement a message-passing library that has better performance. In this thesis, we will discuss the design and implementation of the MP\_Lite message-passing library on top of M-VIA, which is a modular implementation of the VI architecture for Linux. The goal is to combine the high efficiency of MP\_Lite with the high

performance of M-VIA, and exploit the potential of the VI architecture to provide a low latency and high bandwidth message-passing library for applications. MP\_Lite M-VIA uses two different communication modes to achieve low latency for short messages and high bandwidth for long messages. Also, MP\_Lite M-VIA is the first to implement channel-bonding mechanisms on M-VIA, which provide double or triple the maximum throughput by using two or three Fast Ethernet cards in each machine in a cluster computer.

## 1.4 Organization

In chapter 2, a brief introduction to the message-passing paradigm and its implementations will be presented. We will also investigate some user-level communication protocols. The emphasis is on the MP\_Lite message-passing library and the Virtual Interface Architecture. Chapter 3 will discuss the design and implementation details of MP\_Lite for M-VIA. The experimental results, compared to MPICH, MVICH and TCP will be presented in chapter 4. The discussion of the limitations of MP\_Lite M-VIA, as well as a summary and discussion of future efforts, will be presented in chapter 5.

## 1.5 Other Research Efforts

In addition to the thesis work described here, a generic floating-point data compression library was also designed and implemented as the partial fulfillment of the requirement for the degree. The goals of my effort on this project were to develop the initial prototype for the compression library and fine tune the algorithms for the arbitrary precision routines.

Data compression is an effective way to increase the data transfer bandwidth or storage capacity in high performance computing. In the compression library, we only deal with scientific data: integer and floating-point numbers, single and double precision. The goals of the compression library include (CKS<sup>+</sup>00):

- A fast, robust library for application use.
- Utilize determinable and limited amount of resources.

- Run-time resource configuration.
- Operate on local data structure or distributed data structure (via Global Arrays).
- Portability by avoiding assembly level code.

The compression library is still under development. Currently it provides interfaces to compress and uncompress double precision data using different algorithms and contains several utility functions.

Given an uncompressed buffer, the size of the compressed buffer is dynamically determined and allocated and the handle of the compressed buffer is returned to the user. The handle contains the address of the compressed buffer as well as the header information. The compression algorithms currently implemented are:

- Double precision to single precision.
- Double precision to arbitrary precision.
- Skip lists.
- Double precision to arbitrary precision then using skip list mechanism.

IEEE standard 754 specifies that a double precision number contains one *sign* bit, 11 *exponent* bits, and 52 *mantissa* bits. The exponent has a bias of 1023, thus an exponent of zero means that 1023 is stored in the exponent field. In the algorithm of double precision to arbitrary precision, the user specifies how many exponent and mantissa bits are needed and the algorithm adjusts the numerical representation accordingly. The algorithm must deal with several aspects of compressing the numerical representation of a double precision number:

1. If the number of exponent bits the user specified is not enough to represent the data, the algorithm will automatically increase the number of bits to that required. There is a compression option that can let the algorithm automatically determine how many exponent bits are needed to represent the maximum (or minimum) number in the user data.

2. Some representations are reserved for special values. For example, infinity is represented with an exponent of all ones and a mantissa of all zero. These values should be handled differently.
3. Big endian and little endian have different representations of a floating-point number. So the endian information should be stored in the compression header.

The skip list algorithm represents a number in two components: data value and data index. It eliminates the need to represent a zero value therefore it is usefully for any sparse array or matrix. A variation of the skip list is to count the number of continuously repeated values. One of the requirement of the compression library is that user can modify the compressed data without uncompressing the entire buffer. For a skip list compressed buffer, the modification may lead to size change of the buffer. So an extra buffer is provided in skip list compressed buffer to store small changes. If too many changes are made and lead to the overflow of extra buffer, the current implementation must uncompress and re-compress the entire buffer.

Some application level functions are provided to operate on the compressed buffer: get or put a portion of successive data, gather or scatter data according to an index map, and accumulate data.

## CHAPTER 2. COMMUNICATION WITHIN A CLUSTER

Shared-memory multi-processor machines usually use shared-memory for inter-process communication. For distributed systems, especially in clusters, message-passing is a more common approach. The two most commonly used message-passing standards are PVM (Parallel Virtual Machine) and MPI (Message Passing Interface). MPI does not support some features of PVM such as dynamic process spawning, but it has more flexible collective functions (gather/scatter) and asynchronous send and receive communication capabilities. Some commonly used message-passing libraries are investigated in this chapter. The MP\_Lite message-passing library will be discussed in more detail.

Message-passing libraries are implemented on top of an underlying network protocol. Compared to traditional network protocols, a user-level protocol allows the user to bypass the operating system and access the network device directly, thus providing low latency and better performance. We will investigate several user-level protocols and focus on the Virtual Interface Architecture.

### 2.1 Parallel Virtual Machine

PVM (PVM; Sun90) is one of many message-passing systems that preceded the formation of the MPI standard. It is an integrated set of software packages that allows a heterogeneous collection of computers to be used as a single parallel computer. PVM provides a general programming interface for algorithms, and the underlying infrastructure permits the execution of applications in a virtual computing environment that supports multiple computation models, such as functional parallelism and data parallelism. It provides support for a variety of architectures. The processors involved can be scalar machines, multi-processor machines or



other special processors. The principles of PVM include:

- User-selected running host. The user selects a set of machines to run the application on and can exploit the capability of each specific machine.
- The basic unit of parallelism is a *task*. A task is often but not always a process in the operating system.
- Explicit message-passing model. Message-passing is accomplished by using explicit send and receive commands.
- Heterogeneity and multiprocessor environment support.

A typical execution of a PVM application is a set of one or more sequential programs containing embedded PVM function calls in either the C or FORTRAN language. Each application program or instance of the application corresponds to one task. The compiled and linked binary codes are placed in a location accessible from each machine involved. The user starts one task, which eventually invokes other tasks. Those active tasks exchange messages to complete local computations. The results in each node are finally combined.

## 2.2 Message Passing Interface

MPI (For95), which was first defined in 1992, is a widely accepted standard for writing message-passing programs on multiprocessor machines. The standard provides portability between various architectures and an easy-to-use, consistent interface for application development.

MPI is a library that can be called from C, C++ or FORTRAN programs. It is designed to allow efficient inter-processor communication, reduce memory-to-memory copies and allow the developer to overlap communication and computation. The semantics of the interface is architecture independence and language neutral. Therefore, MPI applications can be developed on and for many platforms and used in a heterogeneous environment. MPI provides reliable communication for the upper layer, so applications do not need to deal with communication failures. MPI guarantees thread-safety for multithreaded programming as well.

MPI describes the syntax and semantics for point-to-point communications, collective communications, group, context and communicator management, process topologies, environment management and profiling interface.

In point-to-point communications, the messages are not *overtaking*. If two sends match one receive or one send matches two receives, the destination node will not receive the second message if the first one is pending. In a single-threaded program, the send and receive are also ordered. There are four communication modes for point-to-point communications:

**Standard mode:** The blocking send and receive are standard mode communication. In this mode, a send can start whether or not a matching receive has been posted and can complete before a matching receive is posted. It is up to the MPI implementation to decide whether outgoing message will be buffered and if the send operation should be blocked. The standard send mode is **non-local**: the completion of the send may depend on the matching receive.

**Buffered mode:** This is similar to standard mode, but it is **local**. If a send operation is executed before a matching receive is posted, the outgoing message will be buffered to allow the send call to complete.

**Synchronous mode:** A send can start whether or not a matching receive is posted, but will complete successful only when the matching receive is posted and has started receiving. This is a **non-local** function.

**Ready mode:** A send may start only when the matching receive is posted. Otherwise an error is returned.

In addition to these blocking communication mechanisms, MPI defines non-blocking communication mechanisms. A non-blocking send initiates the data transfer and returns immediately. A wait function needs to be called to complete the operation. Non-blocking communication mechanisms can use all four communication modes described above.

Collective communications involve a group of processes. The collective operations provided by MPI include: barrier synchronization, broadcast, gather/scatter, global reduction opera-

tions such as sum, max and min and many other variations. Collective functions are typically built upon basic point-to-point communication primitives.

Since version 1.1 of the MPI standard, many efforts have been made to add new functionality. MPI 2.0 was introduced in 1997. Many new features were added such as dynamic process creation, one-side communication and parallel IO (For97).

There are many implementations of the MPI standard. Computer vendors usually have their own MPI implementation optimized for their specific architectures. There are also many implementations that are freely distributed and suitable for a variety of architectures. MPICH and LAM MPI are the two most commonly used implementations.

### 2.2.1 MPICH

MPICH (GLnDS96) is a complete implementation of the standard. The initial implementation was available immediately when the MPI standard was released in 1994. The goal of the MPICH project is to provide a portable, robust and efficient MPI implementation and promote the adoption of the MPI standard. MPICH is essentially a base implementation for parallel computers. MPICH is suitable for a variety of architectures. It supports traditional distributed-memory parallel computers (Intel Paragon, IBM SP, NCube, Cray T3D), shared-memory architectures (SGI Origin, IBM SMP, Compaq ES40) and clusters of workstations running Unix or Windows. MPICH is intended to exploit the capability of specific architectures to obtain high performance communications.

The key for performance and portability in MPICH is the Abstract Device Interface (ADI), which is architecture independent. All MPI functions are implemented using macros and functions that make up the ADI. The ADI layer provides basic send and receive functions and message management. It contains codes for message packetizing, attaching headers, buffer management, queue management and handling heterogeneous environments.

For each different architecture, the ADI is implemented by using an architecture specific low level *channel interface*. The channel interface implements three data transfer protocols: The eager protocol where data is sent to the destination immediately; the rendezvous protocol

where data is sent to the destination only when a matching receive is posted; the get or put protocol where data is read or written directly. The simplicity of the channel interface, which can be as small as five functions, provides a quick way to port MPICH to new architectures.

### 2.2.2 LAM MPI

LAM (Local Area Multicomputer) (BDV94) is a full implementation of MPI and is a programming environment for heterogeneous computers on a network. LAM provides enhanced monitoring and debugging tools, such as a snapshot of a process and message status, to facilitate the message-passing application development.

Each computer runs a LAM daemon, which consists of a nano-kernel and a dozen system processes. The nano-kernel schedules these internal processes and some external processes to provide a communication subsystem for message passing between other LAM daemons. The LAM buffer daemon collects incoming messages and stores outgoing messages for forwarding. LAM MPI has the capability of dynamic process spawning, in which a group of MPI processes can collectively create a new group of processes and a new communicator is established for communication.

### 2.2.3 MP\_Lite

MP\_Lite (TCK01) is a light weight message-passing library designed to streamline the data flow and deliver the maximum performance to applications in a portable and user-friendly manner. The purpose of MP\_Lite is to minimize the overhead of the message-passing layer and deliver as much performance as possible to applications. A full implementation of the MPI standard requires complicated buffering and queue management to provide portability for various architectures and handle situations such as `MPI_ANY_SOURCE` in receive operations, out-of-order messages and byte mismatches between send and receive pairs. The extra buffering and memory copy overhead, as well as the complicated multi-layered programming structure, reduce the communication bandwidth and increase the latency.

MP\_Lite provides a subset of the most commonly used MPI functions, which are enough

for a large number of the parallel codes. The simplicity makes it easy to reduce extra buffering and programming overhead, and thus deliver the maximum performance from the underlying network layer to the application. It is an ideal research tool for studying the performance of message-passing. Below is a listing of MPI commands that MP\_Lite supports.

- Initialization and cleanup

MPI\_Init, MPI\_Comm\_size, MPI\_Comm\_rank, MPI\_Finalize.

- Send and receive functions

MPI\_Send, MPI\_Recv, MPI\_Sendrecv, MPI\_Bsend, MPI\_Isend, MPI\_IRecv, MPI\_Ssend, MPI\_Srecv, MPI\_Wait.

- Collective operations

MPI\_Allreduce, MPI\_Bcast, MPI\_Barrier.

- Timing functions

MPI\_Wtime.

- Cartesian coordinate functions

MPI\_Cart\_create, MPI\_Cart\_coords, MPI\_Cart\_rank, MPI\_Cart\_shift, MPI\_Cart\_get.

The MP\_Lite does not support groups, the use of communicators for creating subgroup and the abstraction of the data types in a heterogeneous environment. It is not appropriate for more complex codes using those features. Below is listing of what is not supported.

- Communicators other than Cartesian grid functions and MPI\_COMM\_WORLD.
- MPI\_File\_ and MPI\_O\_ functions.
- Many variations on the basic communication functions.
- Heterogeneous environments.

Figure 2.1 represents the organization of MP\_Lite. Applications can use the MP\_Lite syntax, which is simpler than standard MPI syntax, or choose to use standard MPI syntax. The MP\_Lite layer has support for all the implemented send/receive functions, collective functions, timing functions, IO functions and Cartesian functions. There are also functions for a variety of other ongoing areas of research.

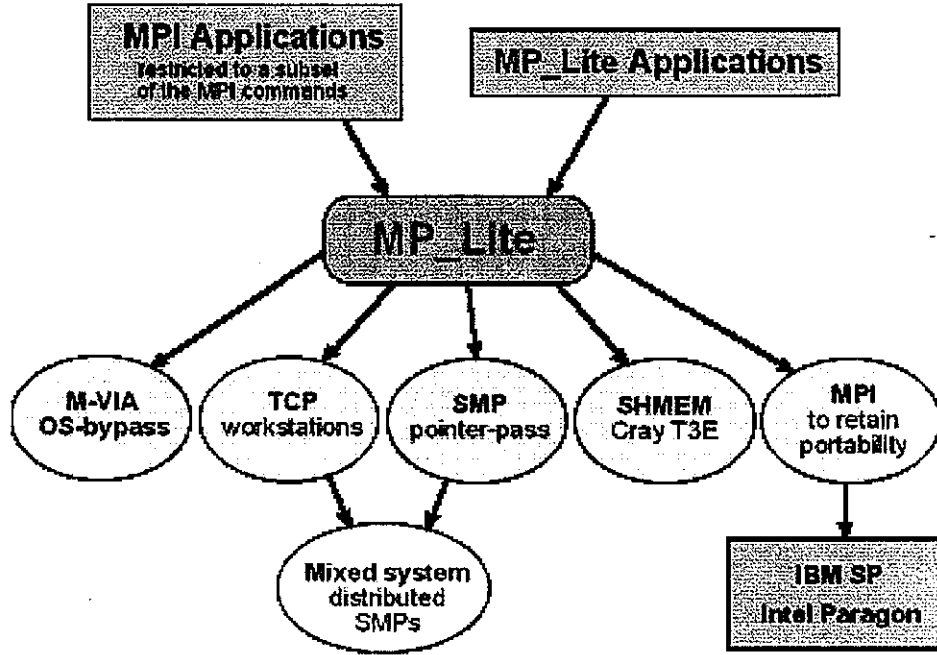


Figure 2.1 Diagram of the structure of MP\_Lite

Each MP\_Lite module implements point-to-point communication functions for different architectures. For the TCP module, there are two modes: synchronous and asynchronous. Communication events within an SMP node can be through either TCP or through shared-memory segments. There is also a module for using the Cray T3E SHMEM library that provides twice the performance of the Cray optimized MPI. In chapter 3 of this thesis, we present the work on the M-VIA module, which bypasses the operating system to provide lower latency and higher bandwidth.

The TCP synchronous module simply increases the TCP send and receive buffers. Because all the messages must be buffered at TCP layer, this is an efficient way to reduce extra buffering and memory-to-memory copies. Therefore, the TCP synchronous module provides the

maximum performance to the application layer. However, it will lock up if the user puts more data than can fit into the enlarged TCP buffers. Setting TCP buffers to a large size can make it usable for many applications but requires large amounts of memory for this configuration.

In the TCP asynchronous mode, the send and receive functions initiate the data transfer but return before completion. Whenever the data is transferred out of the TCP buffer or more data arrives in the TCP receive buffer, a SIGIO signal is generated so that a signal handling routine can continue transferring the data. Asynchronous send and receive functions are non-blocking and are more robust than the synchronous mode. The `MP_Wait()` function will buffer the send data when necessary, therefore it will never be blocked even if two nodes are both sending. This asynchronous mode provides good performance even when using the default TCP buffer size.

All modules of `MP_Lite` implement the basic communication primitives and use the same type of message queues to manage the message buffering when needed, such as for out-of-order messages. They provide a consistent interface to the upper layer though the implementation details may differ.

## 2.3 Other Message-Passing Libraries

In addition to the traditional two-side communication libraries, which require the cooperation of both the source and destination, there are one-sided communication libraries that can put or get messages without the explicit cooperation of the interacting node. The version 2.0 of the MPI standard has some support for one-sided communication, but the typical example is the Cray T3E SHMEM library (SHM94).

There are also several high-level libraries that build upon or beyond traditional message-passing libraries to provide a simple to use interface for applications. As an example, Global Arrays (NHL96) provides a distributed multi-dimensional array interface as well as one-sided communication mechanisms.

## 2.4 User-level Networking

All message-passing libraries are implemented on top of one or more underlying network protocols. The performance of the network protocol is critical to the performance of message-passing libraries. Traditional network protocols such as TCP usually use kernel protocol stacks to handle data transfer and demultiplexing operations. This mechanism requires data being copied multiple times between user space and kernel space. For example, in Linux, to receive a packet, the data is moved from the I/O device to the kernel *sk\_buff* data structure, and then moved to the user buffers. The extra memory-to-memory copy as well as the operating system processing overhead increases the data transfer latency and decreases the bandwidth (CJRS89).

In order to improve the performance, it is desirable to move the network interface much closer to the application. A User-level Networking (ULN) protocol defines an interface between applications and underlying network devices. Applications can talk directly to the network interface controllers through a protected environment, thus reducing the operating system processing overhead and eliminating the extra memory copies. Examples of ULN are U-Net, Active Messages, Fast Messages, Virtual Memory-Mapped Communication, Basic Interface for Parallelism, Scheduled Transfer Protocol and the Virtual Interface Architecture.

### 2.4.1 Active Messages

Active Messages (AM) (vECgS92) is an asynchronous communication mechanism intended to overlap communication and computation. The traditional send/receive model often uses blocking or a handshaking mechanism to implement the blocking communications, and a buffering mechanism to implement the non-blocking asynchronous communication mode. Thus the effectiveness of an application using the message-passing library is degraded under the traditional send/receive model due to poor overlap of communication and computation. In Active Messages, each message contains as its header the address of a user-level handler which is executed on message arrival at the destination side. The handler is executed to extract the message body from the network, which is viewed as a pipeline. The sender launches the mes-



sage into the network and continues computing; the receiver is notified or interrupted on the message arrival and runs the handler to receive the message body. The Active Messages differ from the Remote Procedure Call (RPC) in that the handler executed on the message arrival is to extract the message body from the network instead of performing computation. Buffering is not needed for Active Messages.

### 2.4.2 U-Net

The User-Level Network Interface (U-Net) (vEBBV95) communication architecture provides processes with a virtual view of a network interface to enable user-level access to high-speed communication devices. It focuses on reducing the processing overhead to provide low-latency communication and exploit the full network bandwidth even for small messages. It is an architecture designed to support traditional network protocols such as TCP/IP, as well as newer networking abstractions such as Active Messages.

The U-Net architecture consists of three parts: the *end-point* represents a handler to the network, the *communication segments* hold the communication data and the *message queues* hold descriptors for incoming or outgoing messages. To send a message, the send descriptor is pushed to the send queue and then the network interface will complete the descriptor. Incoming messages are demultiplexed into the appropriate destination based on message tags. The U-Net architecture specifies two levels of communication: a *base-level* which requires an intermediate memory copy at both the source and destination, and a *direct-access* mode which supports true zero-copy data transfers.

### 2.4.3 Fast Messages

Fast Messages (FM) (PKC97) is a low-level messaging layer similar to Active Messages, but expands Active Messages by imposing stronger reliability guarantees. It uses essentially the same API as Active Messages and has the same concept of message handlers, but provides a guarantee for reliable delivery, ordered delivery and control over the scheduling of the communication work (decoupling), which is a mechanism to allow programs to control their

cache performance. This allows the higher message layers the ability to avoid flow control, retransmission and other reliability issues.

#### 2.4.4 Virtual Memory-Mapped Communication

Virtual Memory-Mapped Communication (VMMC) (DBL<sup>+</sup>97) is a communication model providing direct data transfer between the virtual address space of the sender and receiver. The receiver *exports* the destination memory region, and the sending process *imports* remote buffers. VMMC protects the memory access by restricting the exporting and importing of the buffers. After a successful import, the sender can transfer data from its virtual address space into the imported destination buffer. This is accomplished by using a Remote Direct Memory Access (RDMA) mechanism.

#### 2.4.5 Basic Interface for Parallelism

The Basic Interface for Parallelism (BIP) (PT98) is a small API implemented on Myrinet network hardware. It implements all communication in a user layer library and gives the user direct access to the hardware. Memory copies are minimized during data transfer. Short messages are stored in an circular queue, so that send calls will not block even if no matching receive has been posted. Sending a long message requires a receive to be posted before or no longer than 50ms after the send.

#### 2.4.6 Scheduled Transfer Protocol

The Scheduled Transfer Protocol (STP) (fIT00; SGI) is an ANSI specified connection-oriented data transfer protocol. The protocol supports flow-controlled *Read* and *Write* sequences and non-flow-controlled, persistent-memory *Put*, *Get* and *FetchOp* sequences. The objective of STP is to provide high-bandwidth data transfer with minimal host CPU usage for long messages, and very low latency for short messages. STP has been implemented on Gigabyte System Network (GSN) and Gigabit Ethernet for Irix 6.5. The implementation on Linux is under development.

The STP flow-controlled *Read* and *Write* sequences are designed to increase the bandwidth of the long message transfer. A small control message is used to pre-allocate buffers on the destination node, and the user buffers are mapped into the network interface's address space. Therefore, data can be transferred directly from the source user buffers to the destination user buffers using a RDMA mechanism to achieve potentially true zero-copy data transfer.

The non-flow-controlled Get/Put/FetchOp sequences are designed for short messages where low latency is the key. These sequences rely on more persistent memory mapping of the data buffers. The data buffers, once mapped through the kernel, are subsequently used and re-used to send/receive multiple blocks of data several times, thus resulting in very low latencies.

STP provides the basic transport layer infrastructure that can be used to implement multiple Upper Layer Protocols (ULP). Currently the only ULP implemented for Linux is the INET sockets API, AF\_INET sockets of type SOCK\_SEQPACKET using the protocol family IPPROTO\_STP. STP can use hardware acceleration, or use full software support. The current Linux implementation includes a full software support module and the enhancements to Gigabit Ethernet drivers with the Alteon firmware.

The current Linux implementation consists of a complete STP stack for long message transfers through the socket API. However, it does not support reliable data delivery, and is still in a very unstable beta stage. There is also an OS-bypass library (libST) for short message transfers, but it does not work because the receive ring in the device driver has not been implemented and the send sequences can only send header information. Therefore, we have not implemented MP\_Lite for STP.

#### 2.4.7 Virtual Interface Architecture

Virtual Interface Architecture (VIA) (CCC97; DRM<sup>+</sup>98), which is a standard proposed by Compaq, Intel and Microsoft, is an architecture for the interface between high performance network hardware and computer systems. The VIA is designed to enable applications to communicate over a System Area Network (SAN). A SAN is a type of network that provides high bandwidth, low latency communication, and has very low error rates. Very similar to

U-Net, VIA defines a set of functions, data structures, and associated semantics, and provides direct access to the network interface for moving data directly into and out of process memory without additional copies of data and bypassing the operating system in a fully protected manner.

The VIA model consists of several components, as illustrated in Figure 2.2. The application and VI user agent form the **VI consumer** part of VIA. The VI user agent, typically the VI Provider Library, is an API for the application to access the kernel agent and the virtual interfaces. The kernel agent, which is a privileged part of the operating system and usually a device driver, performs operations such as memory registration, and opening/closing network interfaces. The data transfer is through Virtual Interfaces. A network interface controller (NIC) can be associated with multiple Virtual Interfaces. Each VI represents an end-point of a connection. The kernel agent, Network Interface Controllers and Virtual Interfaces form the **VI provider** part of the architecture.

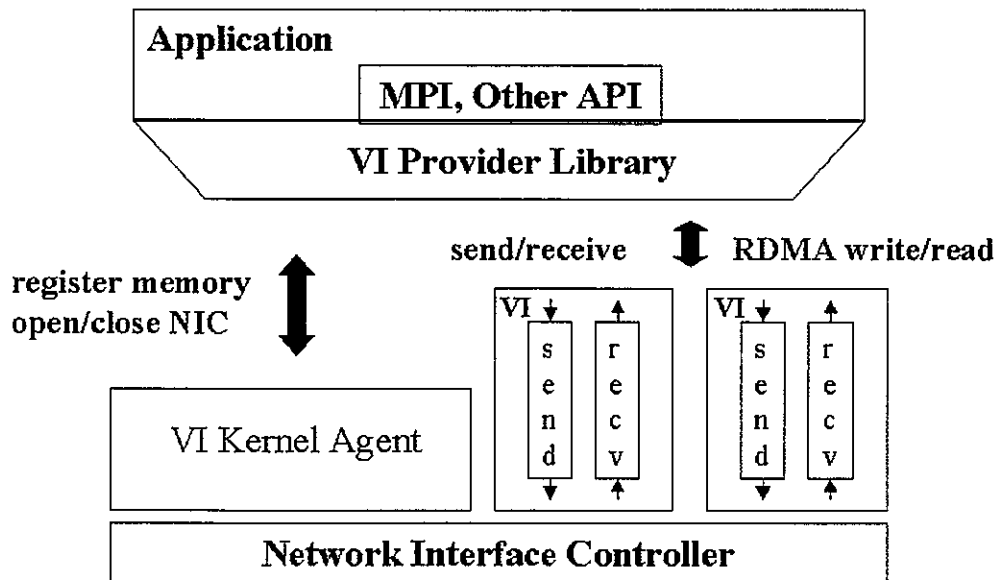


Figure 2.2 VI architecture model

A VI consists of a pair of work queues: a send queue and a receive queue. The VI consumer performs the send and receive operations by posting descriptors to the send queue and receive queue. A descriptor is a data structure that contains all the information that the VI provider

needs to process the requests, such as pointers to data buffers. Each queue is associated with a doorbell. Whenever a new descriptor is posted to the queue, the doorbell is used to notify the underlying NIC. The status information is returned from NIC to the VI consumer. Figure 2.3 shows a diagram of a Virtual Interface.

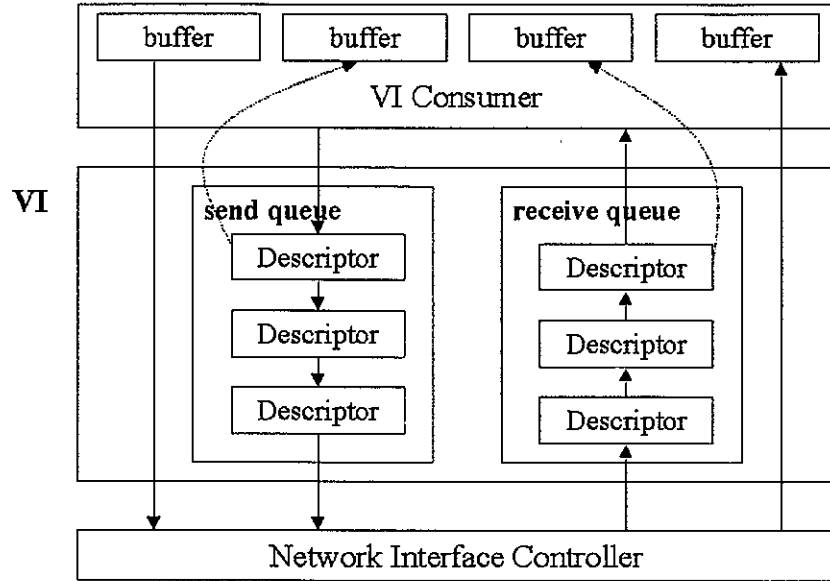


Figure 2.3 A Virtual Interface

Each work queue in the VI can associate with a completion queue. The notification of the completed descriptor in the work queue can be directed to the completion queue. A completion queue allows a VI consume to coalesce notification of descriptor completions from the work queues of multiple VIs in a single location. There are four methods to check the status of a descriptor:

- Poll the send or receive queue.
- Wait on the send or receive queue.
- Poll the completion queue.
- Wait on the completion queue.

The polling method provides the minimum latency but requires more CPU cycles. The VI

specification recommends using completion queues. Waiting on the completion queue is more efficient.

The VI architecture requires that user buffers be registered before they are used. The registration of a buffer locks the buffer memory pages into physical memory and translates the virtual address to a physical address. This memory registration process allows the VI consumer to reuse the registered buffers. The VI provider can transfer data directly between buffers of VI consumers and the network interface controller without additional buffering.

There are two data transfer models in VI: the send/receive model and RDMA model. In the send/receive model, descriptors are posted to the send queue and receive queue. Data is transferred from the buffers specified by the send descriptors to the buffers specified by the receive descriptors. Send descriptors and receive descriptors keep a strict one-to-one mapping and are queued and dequeued in FIFO order. The VI consumer is responsible for the management of flow control, so the receive side must pre-post at least one descriptor of sufficient buffer size before the data arrives.

In the RDMA model, the initiator of the data transfer specifies the address of both the source buffer and the destination buffer. There are two types of RDMA operations: the RDMA Write and RDMA Read. In an RDMA Write, the data is transferred from the local buffer to the remote buffer. In an RDMA Read, the data is transferred from the remote buffer to the local buffer. Prior to the data transfer, the remote VI informs the local VI of the address and the registered memory handle of the remote buffer. The RDMA mode does not consume any descriptors in the remote VI queues, and no notification is given to the remote VI unless the *Immediate Data* field is specified in the local descriptor. The support for RDMA Write is mandatory, while the support for RDMA Read is optional.

The VI architecture supports three reliability levels: unreliable delivery, reliable delivery and reliable reception. All VI NICs are required to support Unreliable Delivery. Other levels are highly recommended but not required. The detailed information about reliability is discussed in chapter 5.

The VI architecture and the Scheduled Transfer Protocol are very similar. They both

provide RDMA mechanisms to increase the bandwidth of long message transfers and use pre-registered buffers for short messages to reduce latency. In STP, the sending of control messages to pre-allocate buffers at the destination is automatically handled by the protocol, but in VIA, the handshaking and the flow-control must be handled by the VI consumer.

## 2.5 VIA Implementations

### 2.5.1 M-VIA

M-VIA (BS99) is a modular implementation of the VIA for Linux being developed by National Energy Research Scientific Computing Center (NERSC) at Lawrence Berkeley National Laboratory. The modular implementation allows it to support many types of network interfaces, and provides a portable and robust interface conforming to the VIA standard. M-VIA consists of a *user provider library*, a loadable kernel agent module and several modified device drivers. It can operate in either hardware acceleration mode or full software mode. It supports hardware VIA “doorbell” or software “doorbell” modes with a fast trap (a trap to a privileged mode that does not incur the overhead of a system call) for legacy hardware.

M-VIA is a full featured implementation of the VIA. The M-VIA kernel module is divided into several independent components including connection management, protection tag management, registered memory management, completion queue management, error queue management and the requisite Linux kernel extensions. The modular design makes it easy to be integrated into current Linux systems, for either the 2.2 or 2.4 kernels of Linux. The hardware support includes: Loop-back driver, DEC tulip Fast Ethernet cards, Intel Pro/100 Fast Ethernet cards, 3Com “Boomerang” Fast Ethernet cards, PacketEngines GNIC-I Yellowfin Gigabit Ethernet cards, PacketEngines GNIC-II Hamachi Gigabit Ethernet cards, Syskonnect SK-98XX Gigabit Ethernet cards and Intel Pro/1000 Gigabit Ethernet cards.

The current release does not have full support for reliable reception. Version 1.2b2 supports reliable delivery, which is very close to the reliable reception level. The latter is required to provide a full robust message-passing library for scientific applications.

The design of M-VIA 2 has been initiated. M-VIA 2 will redesign the internal structure to

provide better support for VI-aware hardware.

### **2.5.2 The Berkeley VIA Implementation**

The Berkeley VIA implementation (BGC98) is a prototype implementation on Sun Solaris, Windows NT and PC Linux over Myrinet. It follows the suggested reference implementation contained in the appendices of the VIA specification. One design choice was to keep as little information in the NIC's memory as possible. The VI creation and connection are protected by mapping a queue for protected commands into the kernel driver's memory so only the kernel driver can perform those operations. Doorbells are implemented as a single memory location on the NIC and polled by the firmware.

The Berkeley VIA implementation only supports a subset of VIA rather than the entire standard. It does not implement the scatter/gather capability, reliability modes, error and completion queues and the RDMA facilities.

### **2.5.3 Commercial Products**

Many vendors provide VI-aware hardware and corresponding VIA implementations. They are: Giganet (Emulex) - cLan, Finisar - Fibre Channel VI Host Bus Adapter, Tandem - ServerNet II, Fujitsu System Technologies - Synfinity CLUSTER, and NEC - V1000 NIC.

## **2.6 VIA Implementations for MPI**

### **2.6.1 MVICH**

MVICH (Cen) is an MPICH-based implementation of MPI over VIA. It provides a high performance MPI for high speed networks such as Gigabit Ethernet, GigaNet, ServerNet II, or Fast Ethernet.

MVICH is a full implementation of the ADI2 for VIA, developed from scratch. It implements four protocols to maximize performance over a range of message sizes:

- For short messages, MVICH uses an eager protocol, in which data is sent and received through pre-posted buffers, with the source sending data immediately.



- For long messages, MVICH uses one of three protocols, depending on whether the underlying NICs support RDMA Write or Read.
1. The “r3” protocol is a standard rendezvous protocol in which data is sent only when the receiver has sent an *ok-to-send* message.
  2. The “rput” protocol is an RDMA Write protocol. Data is sent after an *ok-to-send* is posted by the receiver. Memory on both the sender and receiver is dynamically registered so this protocol is zero-copy.
  3. The “rget” protocol is an RDMA Read protocol similar to “rput”.

MVICH is still under development. The current release is 1.0a6.1. Work is in progress to pass the full conformance and stress tests. We will compare the performance of MVICH with communication libraries in chapter 4.

### 2.6.2 M-VIA for LAM MPI

The ParMa2 project has a basic M-VIA implementation for LAM MPI (BBCR: aToP). It also utilizes the normal send/receive and RDMA mechanism to improve the performance. The basic communication functions supported include:

- Standard send, synchronous send, buffered send and ready send.
- Non-blocking primitives.
- Tag and communicator control on messages.
- MPI.Probe and non-blocking MPI.IProbe, used to read a matching envelope.
- Support for receive from any process: `MPI_ANY_SOURCE` in receive functions.

This package also has a flow control functionality to avoid exhausting all communication resources including RDMA space and pre-posted descriptors. Packet fragmentation and re-assembly are implemented due to the 32 KB limitation of the maximum packet size.

The drawback of this implementation is that user buffers are not dynamically registered. Data is transferred between pre-registered send and receive buffers. Therefore, a memory copy is needed to copy data between the user buffer and the pre-registered buffer at both the source and destination. This greatly reduces the performance for large messages. Moreover, it is currently very unstable. One problem is that it is unable to send messages more than approximately 1600 times. Therefore, it is impossible to run a full NetPIPE (SMG97) benchmark test.

### **2.6.3 VIA for MPI/PRO**

MPI/Pro (DS98; DS99) is a commercial MPI implementation by MPI Software Technology Inc. MPI/Pro uses a progress thread in each of its VI and SMP communication devices for implementing an independent, non-polling message progression, thus MPI/Pro makes progress on all messages independent of the sequence of user calls. Similar to other implementations, two different protocols are used to handle short message send/receive and long message RDMA to achieve the required low latency and high bandwidth. Other features include multiple receive queues and optimized derived data types. Currently MPI/Pro VIA supports Giganet, ServerNet-II and FC-VI (Finisar). The support for Myrinet is in development.

### **2.6.4 MPI Implementation on the NTSC VIA cluster**

The National Center for Supercomputing Applications (NCSA) has implemented a Fast Messages layer on top of VIA for their large scale Windows NT Super Cluster (NTSC), so that MPI-FM, which is derived from MPICH that uses Fast Messages Interface, can run on top of VIA through the Fast Message layer (Pan).

### **2.6.5 MP\_Lite M-VIA**

In the next chapter, we will discuss the implementation of MP\_Lite on top of M-VIA. By combining the light weight, highly efficient MP\_Lite with high performance M-VIA, we will be able to deliver most of the available performance that the underlying hardware offers to the application layer.

## CHAPTER 3. IMPLEMENTATION OF MP\_LITE FOR M-VIA

Using M-VIA to implement message-passing libraries has several advantages. For slower networks such as Fast Ethernet, M-VIA provides much lower latency. For faster networks such as Gigabit Ethernet, M-VIA offers much higher throughput because memory-to-memory copies are minimized. M-VIA can use hardware acceleration to further improve the performance. By combining the light-weight MP\_Lite with M-VIA, we will be able to fully utilize the benefits of both in order to deliver low latency and high bandwidth communication to applications in a portable manner. The goals of this research project are:

- High performance (low latency, high throughput and low CPU load). The MP\_Lite M-VIA module will deliver almost all the performance that M-VIA can offer to the application layer in an optimal situation.
- Channel-bonding capability. MP\_Lite M-VIA will have the capability to use multiple network interface controllers simultaneously to improve potential bandwidth.
- Minimizing resource usage. MP\_Lite should minimize memory utilization and CPU workload. This is important for scalability.
- User friendly. Reduce M-VIA related configuration for MP\_Lite and provide the same interface and configuration mechanisms as other MP\_Lite modules.

### 3.1 System Overview

The MP\_Lite library already provides the high level functions that are independent of the underlying communication protocols. These include global reduction functions and gather/scatter functions. Therefore, what is required for a module is the implementation of the point-to-point

functions, buffer management; message management, queue control, data segmentation and assembly, as well as initialization and finalization procedures. The components of the system and their respective relationships are shown in Figure 3.1.

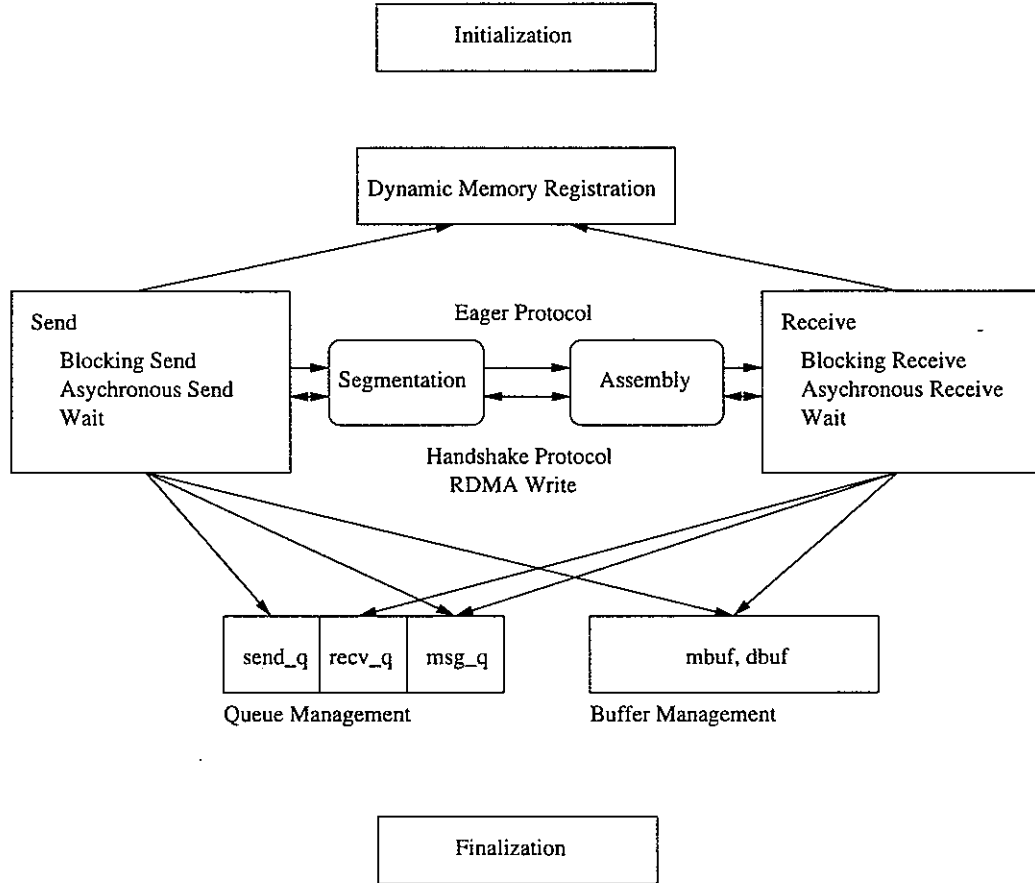


Figure 3.1 MP\_Lite M-VIA module overview

The initialization procedure checks input parameters, allocates memory and sets up connections. The point-to-point functions include blocking and non-blocking asynchronous send and receive commands using two different transmission protocols: the eager protocol and the handshake protocol. Dynamic memory registration is critical for the performance of long message transfers. Data segmentation and assembly is necessary during transmission because of the 32 KB limit of the maximum transfer unit in M-VIA. It is also imperative since we need to use multiple network interface controllers for channel-bonding. The important data structures

in message queue management are the receive queue, send queue and message queue. Buffer management controls the memory resource usage. The finalization stage frees the allocated memory and shuts down related processes.

In the following section, the details of the module implementation of M-VIA for each of the sub-modules are delineated.

### 3.2 Queue Management

Queue management provides a mechanism to buffer and access outstanding messages. The *send* and *receive* queues are used to manage the asynchronous messages. The *message* queue is used to buffer incoming messages that do not have a matching receive. Messages are queued and dequeued in First In First Out (FIFO) order. The related data structures are:

**struct MP\_msg\_entry:** The MP\_Lite message data structure which contains all the necessary information for a message, such as the message id, source, destination, buffer address, length, tag and segmentation information for channel-bonding.

**struct MP\_msg\_entry \*send\_q[]:** Each node has a send queue for all other destination nodes. A message of destination *dest* is appended to the end of *send\_q[dest]*. The send function dequeues messages from the head of *send\_q[dest]* as it delivers the message to the destination node.

**struct MP\_msg\_entry \*recv\_q[]:** Each node has a receive queue for each source nodes. Messages expected from source *src* are posted to the end of *recv\_q[src]*. When a message is coming from *src*, the *recv\_q[src]* is searched from the beginning for a match. *recv\_q[-1]* is reserved for messages whose source is a wildcard.

**struct MP\_msg\_entry \*msg\_q[]:** The buffered message queue is for incoming messages that do not have a match in *recv\_q*. A message that is sent to itself is also posted to *msg\_q*.

The separation of queues by message destination or source speeds up the demultiplexing of incoming and outgoing messages which enhances the performance. An example of the *recv\_q* is shown in Figure 3.2

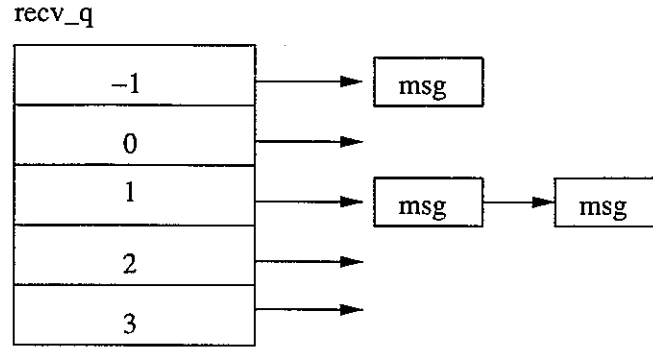


Figure 3.2 An example of the receive queue

Functions related to queue operation are:

**post():** Post a message to the send\_q, recv\_q or msg\_q

**send\_to\_q():** Send a message directly to msg\_q, which is used only when a node sends a message to itself.

**recv\_from\_q():** Try to retrieve a message from the msg\_q. This is the first step to receive any message. When a match is found, the data is copied to the destination buffer and the message in msg\_q is dequeued and destroyed. A receive message matches if the tags of these two messages are the same or the tag of the receive message is a wildcard and the number of bytes is less than or equal to the expected length.

**find\_a\_posted\_receive():** Find a matching receive in recv\_q when a message is coming. If a match is found, the message is returned and dequeued from the recv\_q.

### 3.3 Buffer Management

Sending and receiving is accomplished by posting descriptors, which describe the data address, length and registered memory handle. The short messages are copied to the pre-registered buffers for sending (long messages use user buffers directly). Receive descriptors need to be pre-allocated before connection is setup in order to receive unexpected data before user buffers are available. Because limited memory resources, buffer management is needed

to control the memory usage. In our implementation, we use the concept of *mbuf*, which is similar to the data structure used in many operating system memory management designs. A *mbuf* is a block of memory that contains both the buffer description (in our case, the VI descriptor) and the actual buffer space. An *mbuf* is linked as a queue. Functions are provided to queue and dequeue a block of *mbuf* from the head of the queue. An *mbufs* is allocated in a contiguous address space so that when it is registered, we get only one memory handle to make things easier. This is illustrated in Figure 3.3.

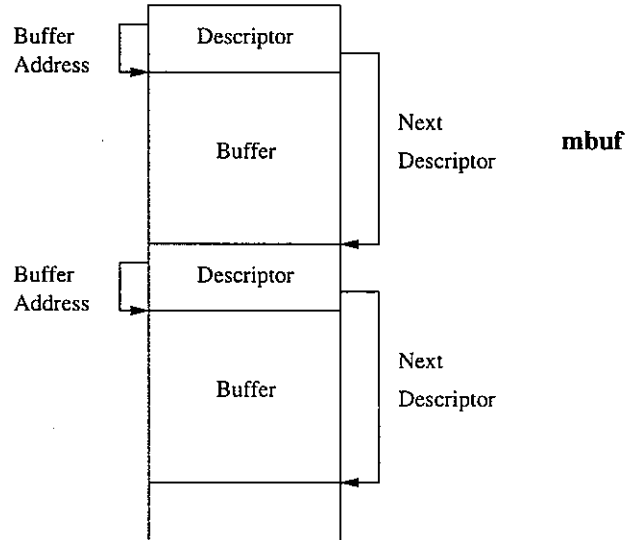


Figure 3.3 mbufs

In addition to *mbuf*, there is another type of buffer unit called *dbuf*. A block of *dbuf* only contains a VI descriptor and does not have its own buffer space. An *mbuf* is for sending and receiving small messages, which are always be buffered in *mbufs* before sending or receiving. A *dbuf* is for sending large messages. The buffer pointer will be redirected to the actual user buffer. The advantages of the separation of *mbuf* and *dbuf* are:

1. Because the size of *dbuf* is small, we can allocate a lot of *dbufs* for sending large messages without greatly increasing the system resource utilization. For example, we can allocate 300 *dbufs* (descriptors) for sending messages of up to 8 MB (each descriptor can point to a 32 KB block of user data).

2. We can increase the size of *mbuf* to improve the short message performance. Because an *mbuf* is only used for sending small messages, which do not require many descriptors, we can increase the size of an *mbuf* without greatly increasing the total system memory usage. For example, we can set the size of an *mbuf* to 16 KB, so that a message smaller than 16 KB can be sent in one descriptor.

In the MVICH implementation, there is only one type of buffer *vbuf*, which is similar to *mbuf*. An *vbuf* is used to send both small and large messages. To send a large message, lots of *vbufs* are needed. Because each *vbuf* has its own buffer space, to reduce resource usage, the *vbuf* size should be small. For example, set the *vbuf* size to 1 KB in MVICH. A message of size 5 KB needs to be send 5 times, which limits the MVICH performance.

Functions related to *mbuf* (*dbuf* is similar) are:

**via\_desc\_request():** In response to the user buffer request, dequeue a block of *mbuf* from the *mbuf* list for usage.

**via\_desc\_release():** When finished using an *mbuf*, queue the *mbuf* to make it available again.

**via\_desc\_restore():** Restore the default value of the descriptor in an *mbuf*.

### 3.4 Important Data Structures

**struct via\_conn:** This is the data structure represents the VIA connection. All the information of a VI connection, such as the VI handle, the connection handle and the remote address, is included in this data structure. Since the current M-VIA implementation does not provide fully reliable data transfer, a *sending sequence number* and an *expected receiving sequence number* are added to improve the error detection.

**Message headers:** Message headers tell the destination what type of incoming message it is. They can be used to distinguish messages and selectively receive them. They are also called message envelopes. To reduce transfer overhead, we use variable size headers instead of a large fixed one to keep the header as small as possible. A few fields of the



beginning of these headers are identical, so they have a common small header for easy analysis. There are four types of headers used in different transmission modes:

1. OP\_SEND: Normal send by using the eager protocol. Data is accompanied with the header. The message length and tag are included in the header.
2. OP\_RDMAW\_RTS: RDMA Write *request-to-send*. Parameters include message length, tag and source message id.
3. OP\_RDMAW\_CTS: RDMA Write *clear-to-send*. Parameters include destination buffer length, tag, source message id, destination message id, and registered destination memory handle.
4. OP\_RDMAW\_DONE: This is used to notify the data destination that an RDMA Write operation is done. This header contains the message length, tag, destination message id and destination memory handle. This header can be eliminated if using the *ImmediateData* field of the descriptor to inform the completion of the RDMA operation.

### 3.5 Initialization

Initialization is done in the `MP_Init()` function. The library needs to read and analyze input arguments, determine the process id, initialize log and status files, allocate and create data structures and setup VI connections.

The run-time parameters are stored in a configuration file `.mplite.config` in the current working directory. The configuration file is created by the `mprun` startup script. The format of this file is:

```
<number of nodes>
<number of NICs>
<program name and arguments>
0 <node0 NIC0>, <node0 NIC1>, ...
1 <node1 NIC0>, <node1 NIC1>, ...
```

...

Each node started by *mprun* reads those parameters and begins to determine its own process id. The process id is an integer starting at zero and uniquely identifies each node. Because multiple nodes can run on the same machine (especially on an SMP machine), and they are basically identical, we need a mechanism to avoid contention in determining the process id. It would be easy if the *mprun* script could determine the process id when it launches each process, and then transfer this id as an input argument to each process. However, because Fortran support for command line arguments is limited, it is not easy to deliver the process id to the correct process if multiple identical processes are running on the same machine. So each process has to determine the id independently.

Our approach is to use System V shared memory to determine the unique process id. All the nodes on the same machine try to create a named shared memory region. The name of the shared memory region is unique to each *mprun* session. If the shared memory region already exists, then the processes try to attach to this memory region. The shared memory region contains an integer. The initial value of this integer is zero. Each process grabs the current value in the shared memory region and increments the value by one. Of course the shared memory needs to be locked using a semaphore to avoid contention from other processes accessing the same shared memory region. All the processes running on the same machine will get different values and can be ordered accordingly. Each process uses the grabbed value combined with the value read from the file *.mplite.config* to determining its unique process id. The last process closes the shared memory region.

After determining the unique process id, the next step is to determine the network devices to be used (the VIA device name), such as `"/dev/via_eth0"` for the first NIC, `"/dev/via_eth1"` for the second NIC, etc. The NIC name or IP address must be translated to the specific device name. In MVICH, the device name is fixed in the source code, so if you want to use another NIC on your machine instead of the default one, you have to recompile the MVICH package. The M-VIA implementation of LAM MPI uses a configuration file to store the VIA device

names, so you have to manually modify the configuration file if you want to use another NIC<sup>1</sup>.

MP\_Lite can stripe data across multiple NICs simultaneously to increase the transmission bandwidth. The MP\_Lite implementation dynamically determines the device from the user provided NIC name at run-time. It works by getting the IP address of the specified NIC name, using the *ioctl()* function to get a list of all the network interfaces installed on the system and comparing the IP address with each of these interfaces. Dynamic configuration eliminates the need for special configuration options for the M-VIA module and keeps the arguments of *mprun* the same as for other modules.

The VI initialization procedure also allocates memory and creates data structures. This includes allocating all the message and queue structures, allocating and registering *mbufs* and *dbufs* (whose address must be properly aligned for performance), opening the VI devices and creating VIs.

The last step of the MP\_Lite initialization stage is to set up a fully-connected network. Connections must be made between each pair of nodes. Each VI can only represent one connection, so we have to create  $nprocs - 1$  VIs and make  $nprocs - 1$  connections in each node for  $nprocs$  nodes. Each VI is given the local and remote address when created. The discriminator (similar to the port number in TCP, but not restricted to integers) of each VI is specified as a triplet {local node id, remote node id, NIC id}. Thus different VIs on the same node have different discriminators.

The connection sequence is determined by the process ids. Each node accepts a connection from nodes with a smaller id, then each node initiates connections to nodes with larger id values. To synchronize this procedure, every node will send a *go* signal to its upper neighbor and receive a *go* signal from its lower neighbor after all connections are generated.

### 3.6 Communication Protocols

Two communication protocols have been implemented in the MP\_Lite M-VIA module: the **eager protocol** and the **handshake protocol**. The eager protocol is for short message, and

---

<sup>1</sup>In fact, due to at least one bug, you can only use the first NIC unless you utilize some non-trivial hacks.

the handshake protocol is for long messages.

### 3.6.1 The Eager Protocol

The eager protocol assumes the receive node has enough pre-posted buffers to hold the incoming messages. Once messages are posted for sending, messages accompanied with headers are sent to the destination node immediately. On the destination node, the arriving messages are stored in the pre-posted buffers and copied to the user buffers when a matching receive is posted. Because of limited buffer resources, this protocol is only suitable for small messages. The eager protocol is illustrated in figure 3.4.

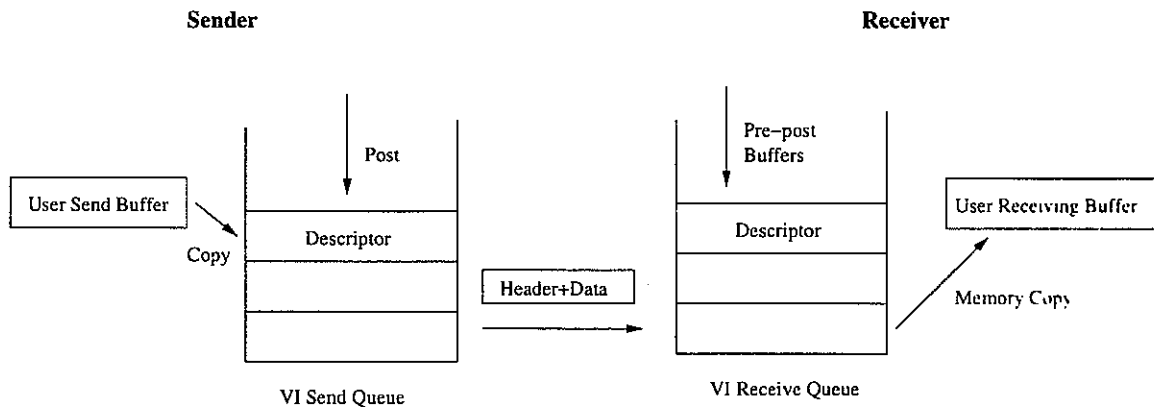


Figure 3.4 Diagram of the eager protocol

The eager protocol can significantly reduce the communication latency since messages are sent without delay. However, it requires pre-posting enough buffers to hold the incoming data from arbitrary sources and at least one memory copy is needed at the destination node to copy data from the pre-posted buffers to the user buffers.

The MP\_Lite M-VIA implementation involves an additional memory copy at the source node, from the user buffers to the pre-registered *mbufs*. A procedure can be implemented that dynamically registers the user buffers and posts the user buffer directly to the VI send queue. However, for small messages, it takes more time to register/deregister buffers than to

copy data to pre-registered buffers. Table 3.1 shows the time comparison of memory copy and registration/deregistration of different data sizes on an Intel PIII PC.

Table 3.1 Memory copy compared to memory registration

Data size (bytes)	Memory copy ( $\mu s$ )	Registration/deregistration ( $\mu s$ )
8	0	4
64	0	4
256	0	4
2048	1	4
4096	2	5
8192	4	6
16384	47	8
32768	91	12
65536	181	21

The table shows that when the data size is less than 8 KB, the memory copy is faster than memory registration/deregistration. Therefore, for small messages, it is more efficient to use the memory copy. For large messages, we switch to the handshake protocol and use the RDMA Write to achieve high performance, zero-copy data transfer.

### 3.6.2 The Handshake Protocol

The handshake protocol requires handshaking between the source and destination nodes. Because of the handshake delay, it is suitable only for large messages. The source node sends out a *request-to-send* control message that includes the message size and tag. When the destination buffer is available, the destination node replies with a *clear-to-send* message containing the destination buffer address and the registered memory handle. The source node then uses an RDMA Write mechanism to deliver data directly into the destination buffer. No extra memory copy is needed<sup>2</sup>. This is illustrated in figure 3.5.

The handshake protocol is more robust than the eager protocol since the source will not send messages until there is enough room at the destination. It can deliver very high bandwidth when combined with the RDMA Write mechanism to achieve a zero memory copy data

<sup>2</sup>In fact, M-VIA still has one internal memory copy at the receive side if no hardware acceleration is available.

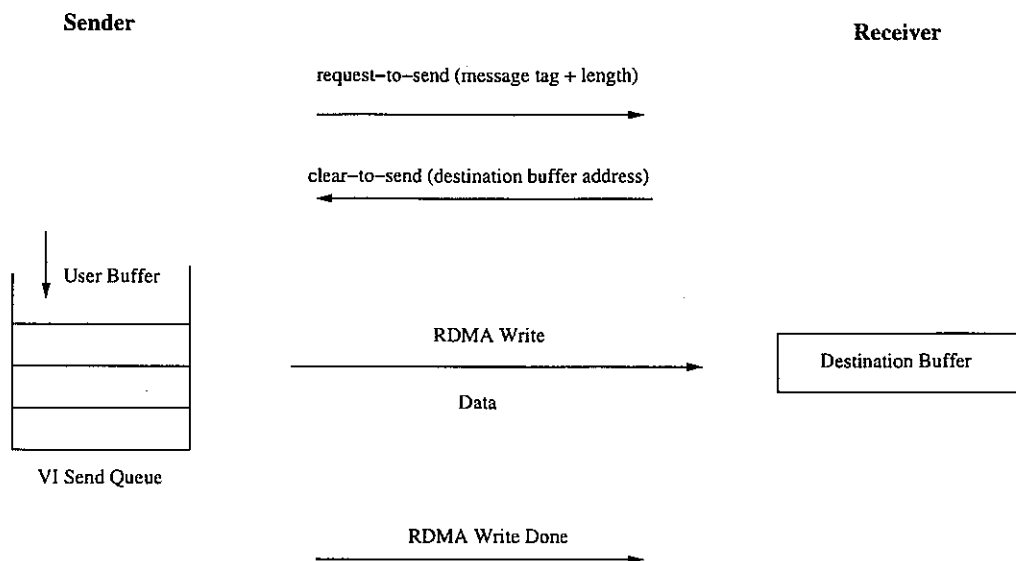


Figure 3.5 Diagram of the handshake protocol

transfers. Although handshaking delays exist, for large messages, the data transmission time is significantly larger than any such delay.

It is possible to devise another simple handshaking protocol that reduces one handshake and can overlap communication and computation at the receive side. Whenever a receive is posted, the destination node just sends out a *clear-to-send* message to the source node, then continues working on other computational components. The source node does not send out any message when a send message is posted. Instead, the source node just waits for a matching *clear-to-send* message, then starts the RDMA Write procedure to send data without the interaction of the destination node. After the message is written remotely, a *RDMAW Done* message is sent to the destination to notify that the transfer is complete.

This method has problems however. Consider what happens if both nodes are going to send. They will both be waiting for *clear-to-send* messages, which leads to deadlock. Another problem exists in channel-bonding. Because the receive node can have a larger buffer than source message, it is unable to determine how to segment the data and register the buffer unless it receives the source buffer length information. Therefore it can not send a *clear-to-send* beforehand.

### 3.7 Dynamic Memory Registration

In the RDMA Write mode, whenever a send or receive is posted, the corresponding buffer is dynamically registered. The registration of a buffer is to pin the buffer into the physical memory. When the data transfer is finished, the buffer is deregistered. The frequent registration and deregistration may decrease performance.

One optimization in MP\_Lite M-VIA is to keep the registration information for the last few registered buffers. When a buffer is registered, the buffer address, length and the registered memory handle are put into a memory registration cache. When the data transfer is completed, the buffer is not deregistered immediately. Instead, the buffer registration information is still stored in the cache. Before registering a buffer, the cache is searched to see whether a registered buffer is available for use. In case of a cache hit, the registration information in the cache can be used immediately, thus eliminating the overhead of memory registration.

There are three statuses of a cache entry: INVALID, CACHED and IN\_USE. An empty cache entry is marked INVALID thus can be used to register a new buffer. If a cache entry is being used by any of the MPI send or receive commands, it is marked IN\_USE. If all of the MPI send or receive commands that use the cache entry are completed, the cache entry is marked CACHED. In case of a cache miss, an empty cache entry is searched first to register the new buffer. If the cache is full, the *least recently used* cache entry is replaced. However, a cache entry that is in IN\_USE status can't be replaced because the data transfer is not completed for this entry. If all the cache entries are in use, then the newly registered buffer will not use the cache.

In M-VIA, there are limitations on the size of buffers and the number of buffers that can be registered. It is necessary to clean the cache if the memory registration will exceed those limitations.

### 3.8 Send

In MP\_Lite, there are two essential send functions: MP\_Send() and MP\_ASend(). MP\_Send() is a blocking function that does not return until the message is received or stored somewhere

so that the send buffer is free for reuse by the sending process. The `MP_ASend()` function is a non-blocking, asynchronous function that returns after the function is called. It only indicates that the sending mechanism has started; it has not completed. The buffer can not be reused until a matching `MP_Wait()` is called. The implementation of `MP_Send()` is just an `MP_ASend()` followed by an `MP_Wait()`.

In `MP_ASend()`, the message destination is checked first. Messages sent to oneself are copied directly to `msg_q`. Other messages are posted to `send_q`. `MP_ASend()` does not actually start sending the message. The actual sending begins only when `MP_Wait()` is called. `MP_Wait()` takes a message from the head of the `send_q` and begins to deliver the message. This message might not be the message that matches the `MP_Wait()` call. The procedure is repeated until the message corresponding to the `MP_Wait()` call is taken out of the `send_q` and has been delivered.

The send is implemented by using two transmission protocols described in the last section, eager protocol and handshake protocol. Small messages are sent using eager protocol. Messages less than 12 KB accompanied with `OP_SEND` headers are copied to the pre-registered *mbufs* and put into the send queue. In the eager protocol, we assume the destination has enough space to store small messages, so in most cases sends will not be blocked. If the destination node does not have enough buffers, data will be lost. In a reliable version of M-VIA, the lost data is supposed to be re-transmitted by M-VIA. In an unreliable version of M-VIA, currently only error messages are generated by `MP_Lite`.

Large messages use the handshake protocol and RDMA Write mechanism. The source node sends an `OP_RDMAW_RTS` (RDMA Write *request-to-send*) message to the destination node with the buffer length and tag being specified in the header. The sender then waits for the `OP_RDMAW_CTS` (RDMA Write *clear-to-send*) message. It is necessary to check the destination buffer length in the reply. If the destination buffer is large enough, then we can begin the RDMA Write session by transferring data from the source buffer directly to the destination buffer. No additional memory copy is needed. If the destination buffer is too small, it is not considered an match. Because of the maximum 32 KB transfer size limits of



M-VIA, for messages larger than 32 KB, we need to segment (e.g., packetize) the data before transmission.

There are two choices when sending large messages. The first one sends out a 32 KB descriptor, waiting for it to complete then using the same descriptor to send another 32 KB of data. This method requires only one descriptor, thus reducing the memory usage. The second approach, which is the default method, posts as many descriptors as necessary to send a message. Although this method requires more descriptors, the throughput is better for Gigabit Ethernet.

### 3.9 Receive

MP\_Lite has two types of receive functions: `MP_Recv()` and `MP_ARecv()`. `MP_Recv()` is a blocking receive function, and `MP_ARecv()` is a non-blocking, asynchronous receive. In the actual implementation, `MP_Recv()` is just a `MP_ARecv()` followed by a blocking `MP_Wait()`. `MP_ARecv()` does nothing other than put the message into the `recv_q`. The `MP_Wait()` handles the actual data transfer.

The receive procedure in `MP_Wait()` starts by checking the `msg_q`. The message might have already been received and buffered in `msg_q`. If a matching message is found, the data is copied from the `msg_q` to the destination buffer and the buffer in `msg_q` is freed. Before copying the data, it is important to wait until the message is completely received. If the receive buffer is larger than the buffered message in `recv_q`, after data is copied to the receive buffer, the search in `msg_q` should be continued to find another match that can fill the available space in the receive buffer.

If the message is not found in `msg_q`, it needs to be actually received over the network. The VI receive function is called to wait on the VI receive queue until a message header is received. By distinguishing different types of message headers, different operations are performed:

- If the header is `OP_SEND`, it is a small message sequence that will use the eager protocol.

The receive side extracts the message length and tag and tries to find a matched receive in the `recv_q`. If the tags of the send and receive message are the same, or if the receive

tag is a wildcard, then they are matched. If no such match is found in the `recv_q`, the incoming message needs to be buffered. This is done by allocating a temporary buffer and creating a new message. After receiving the incoming message in the temporary buffer, the message is posted to the `msg_q`. If a posted receive is found that matches the incoming message, then the incoming message stored in the pre-posted *mbufs* is copied to the destination user buffer.

Things would be easy if all the matched send and receive messages were the same size. If the receive buffer is smaller than the send buffer, it is a mismatch and another posted receive should be searched. In the case where the message sent is smaller than the receive buffer, the message is copied to the receive buffer and the progress of the receive buffer is adjusted. The receive buffer is put in the `recv_q` again. This allows following incoming messages being received into this receive buffer.

- If the header is `OP_RDMAW_RTS`, it is an RDMA Write *request-to-send* message. First `recv_q` is checked to find a matching receive. If no such match is found, a temporary message buffer is created for the incoming data. This buffered message is put in the `msg_q` even though no data has been received. An `OP_RDMAW_CTS` message is sent to allow the RDMA Write to begin. If the receive buffer is larger than the send buffer, after data transfer is completed, the progress of the receive buffer is adjusted and the receive buffer is put in the `recv_q` again.
- If the header is `OP_RDMAW_CTS`, it is an RDMA Write *clear-to-send* message, and is a response to the previous `OP_RDMAW_RTS` request. The message id is extracted from the header to find out which message made the request and the RDMA Write operation is started for this message.
- If the header is `OP_RDMAW_DONE`, it is an acknowledgment from the source node that an RDMA Write operation has been completed. The destination node extracts the message id from the header to know which message has been done, then adjusts the *number of bytes left* field of the message to adjust the current state. It is not necessary

for the entire message to have been received because the receive buffer may be larger than the message sent. The actual implementation uses the *ImmediateData* field of the descriptor to send the last data packet, so that the last packet will consume one descriptor at the destination side to indicate the completion of the data transfer.

The destination node repeatedly receives headers and progresses each receive until the desired message is received. A special case is when the source of the receive message is a wildcard, matching any source. The destination node cycles through each source by using a method outlined above to see whether a matching header has arrived. This is not very efficient, since we must check each source, but it is convenient at this time.

### 3.10 Channel-Bonding

Channel-bonding is the ability to stripe messages across multiple network interface cards to improve the potential bandwidth between machines in PC and workstation clusters. Channel bonding was first introduced in the Beowulf parallel workstation (SSB<sup>+</sup>95), where using two Ethernet channels could sustain 70% or greater throughput than a single network alone.

Compared to channel-bonding on multiple Fast Ethernet cards, using one Gigabit Ethernet card in the same situation does provide higher throughput, but this greatly increases the cost of the whole computer system. Channel-bonding on multiple Fast Ethernet cards provides an economic and scalable way to improve the communication performance in clusters.

To enable channel-bonding, it is necessary to allocate a copy of related data structures such as the NIC handle, VI handle, *mbuf*, and connection descriptor for each NIC. During the initialization stage, a full connection network is constructed for each NIC. That is, NIC 0 on all nodes will form a fully connected network, NIC 1 on all nodes will form a separate fully connected network, etc.

MP\_Lite M-VIA defines a size threshold for starting channel-bonding. Long messages usually can use channel-bonding. For small messages sent by the eager protocol, if the message size is larger than the channel-bonding threshold, the message can also be sent using multiple NICs. The data buffer is divided into blocks of data, where each data segment is stored in one

VI descriptor. For two NICs, a header with the first block of data is posted to the send queue of the first NIC, the second block of the data is posted to the send queue of the second NIC. After both sends have completed, the third block of data, if available, is posted to the send queue of the first NIC again. This procedure continues until all the data is sent. Each NIC sends one descriptor each time, in order. The destination node receives blocks of data from each NIC in the same order, as in figure 3.6.

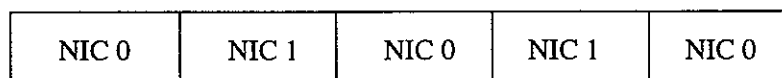


Figure 3.6 Channel-bonding for small messages

For large messages, it is necessary to register buffers used by every NIC, so it is better to divide the buffer into approximately equal length segments with each NIC handling one segment of data. Remember that the destination node needs to reply with the address and the registered memory handle of each segment to the source node. The destination node is unable to do so before the *request-to-send* message is received from the source node, since the receive buffer may have a different size than the send buffer thus the receiver does not know how to segment the buffer using the same mechanism as the source node. But it can be assumed that the size of the receive buffer is equal to the size of the send buffer, so the receive buffer can be registered before the arrival of `OP_RDMAW_RTS`. This improves the performance in most situations. Finally, if the sizes are different, the buffer can be deregistered and re-register using the new buffer size. The segmentation is illustrated in figure 3.7



Figure 3.7 Channel-bonding for large messages sent by the RDMA Write

After a segment has been transferred, the NIC needs to notify the destination node of the completion of the data transfer. Only when all notifications from each NIC have been received,

the data transfer is completed.

### 3.11 Finalization

The finalization step frees all of the resources allocated by the MP\_Lite library. This include disconnecting all of the connections, deregistering and freeing all *mbufs* and *dbufs*, destroying VI data structures, and freeing all other memory allocated by the library. It is necessary to synchronize the execution of each node before cleaning up.

### 3.12 Porting M-VIA to the Alpha Platform

Currently M-VIA is only tested for PC x86 platforms running Linux. We have made some changes to the source code of M-VIA so that it also works on the Alpha Linux.

The first change needed is the doorbell type. The doorbell is an operating system mechanism for a process to notify the VI NIC that a descriptor has been placed on a work queue. Three doorbell types are provided by M-VIA: *fast trap*, *ioctl* and *register*. The register doorbell is not yet implemented in M-VIA. The x86 version uses fast trap. However, the fast trap code is written by using x86 assembly language to bypass the OS system calls. For the Alpha platform, it is necessary to disable the fast trap and use *ioctl* doorbell instead. The *descriptor offset into the physical page* field in the doorbell token format needs to be slightly increased because the page size on Alphas is 8 KB compared to 4 KB on the x86 platform.

Another problem is the mapping among user virtual addresses, kernel virtual addresses (linear address) and physical addresses. In the memory registration function, a user virtual address needs to be mapped to a physical address. This is accomplished in the macro *generic\_virt\_to\_phys()*, which walks through page tables to get the physical address. The result of the page table walking in the current M-VIA implementation is the kernel virtual address on Alpha instead of the physical address. It needs to be further translated to a physical address by adding a PAGE-OFFSET. Also, the input parameter to the kernel function *MAP\_NR()*, gets a memory map index for a page in the kernel memory, should be a kernel virtual address instead of a physical address as in the current M-VIA.

One problem still not solved is the size of the memory handle. It is defined as a 32 bit unsigned integer. However, according to the VI specification and the actual programming, the memory handle is obtained by  $(\text{Virtual Address} \gg \text{PAGE\_SHIFT} - \text{PROTECTION INDEX})$ . On the Alpha platform, the address is 64 bit, so theoretically, a 32 bit memory handle is not enough. In our changes to the source code, we did not increase the size of the memory handle because it is related to many other data structures, and thus a non-trivial aspect of the port. The M-VIA implementation should handle this through a normal abstraction mechanisms and this advice has been sent to the developers.

## CHAPTER 4. PERFORMANCE OF MP\_LITE M-VIA ON LINUX

### 4.1 Experimental Environment

#### 4.1.1 Configuration

The performance evaluation environment consists of two test clusters. The first cluster contains two Pentium III PCs connected back-to-back by multiple Fast Ethernet and Gigabit Ethernet cards. The second test-bed consists of two Compaq DS20 Alpha workstations, also connected by multiple Fast Ethernet and Gigabit Ethernet cards. The configurations of these two clusters are shown in table 4.1.

Table 4.1 Test cluster configuration

	<b>CPU</b>	<b>memory</b>	<b>Fast Ethernet</b>	<b>Gigabit Ethernet</b>
PC cluster	Pentium III 450MHz	256MB	DEC Tulip 3Com 3C59X Intel/Pro 100	Syskonnect Hamachi
Alpha cluster	Compaq DS20 500MHz	1.5GB		Syskonnect

The clusters are running the Red Hat 6.2 Linux distribution with kernel version 2.2.19. The M-VIA version is 1.2b2, which supports reliable delivery. We applied our Alpha patch to this version of M-VIA. In the experiment, three different M-VIA implementations of MPI are compared as shown in table 4.2.

Table 4.2 Installed M-VIA implementation for MPI

<b>MPI software package</b>	<b>M-VIA Implementation</b>
MP_Lite 2.2	M-VIA module
MPICH 1.2.0	MVICH 1.0a6.1
LAM MPI 6.3.2	ParMa2 VIA patch 0.3

### 4.1.2 NetPIPE Performance Evaluator

For all tests we used the NetPIPE (SMG97) performance evaluation tool. NetPIPE stands for the Network Protocol Independent Performance Evaluator. The network performance is evaluated using multiple ping-pong tests. The transfer block size is increased from a single byte until transmission time exceeds one second. The transmission of each size of data block is repeated enough times so that the total time is far greater than the timer resolution. NetPIPE reports the block size in bytes, throughput in Mbps (Megabits per second), and transfer time in microseconds. The latency for a 1-byte message is also reported.

Two types of graphs are presented using the NetPIPE output:

**Throughput graph:** This is the graph of the throughput versus the message size on a logarithmic scale. The throughput graph is the traditional way to show the transfer rate for each different block size. It is easy to see the maximum throughput in this type of graph.

**Signature graph:** The throughput versus the elapsed time on a logarithmic scale. This graph shows the network transfer latency and the network transfer “acceleration”. The latency is the time of the first data point on the graph (1-byte round-trip time divided by 2).

## 4.2 Point-to-Point Communication

In this section, the results of the performance comparison for various communication libraries are presented. The communication is between a pair of Fast Ethernet or Gigabit Ethernet interfaces on one of the test clusters.

### 4.2.1 Fast Ethernet on the PC Cluster

Figure 4.1 shows the throughput comparison of MP\_Lite M-VIA, MVICH, LAM MPI M-VIA, MPICH and raw TCP between Tulip Fast Ethernet cards on two PCs. Raw TCP offers a maximum of 89 Mbps throughput. Both MP\_Lite M-VIA and MVICH can deliver the maximum TCP performance adequately. The maximum throughput of MP\_Lite M-VIA is 91



Mbps, which is a little better than the maximum throughput of TCP. MPICH loses 10% of the TCP performance. The LAM MPI M-VIA has 80% of the TCP performance. For LAM MPI M-VIA, there are stability problems in the current version, so we had to reduce the repeat times when testing using NetPIPE, so the result are a little noiser than other tests.

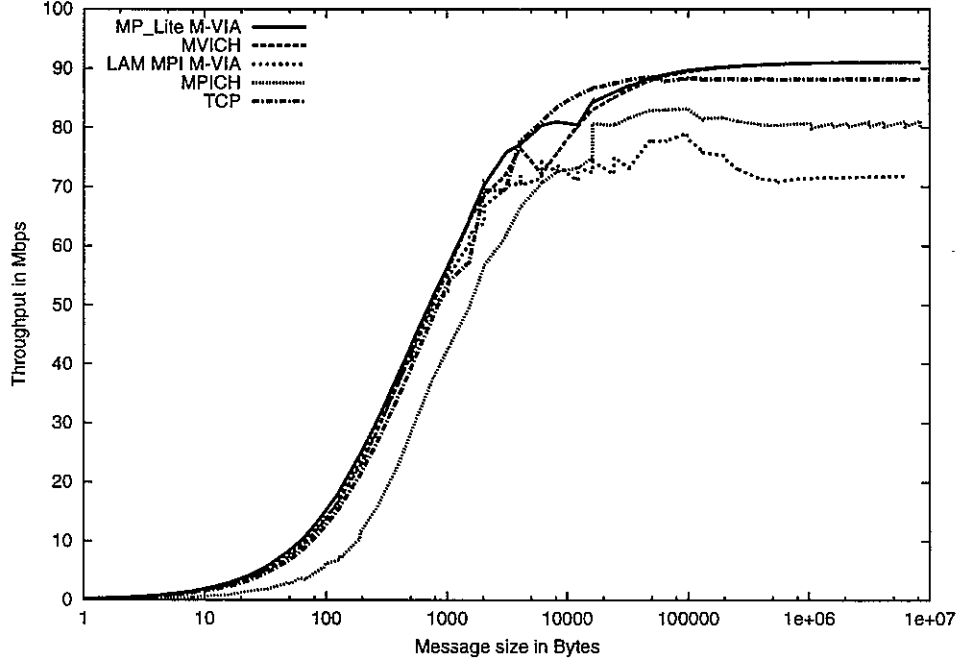


Figure 4.1 The throughput between Tulip Fast Ethernet cards on two PCs

For messages smaller than 8 KB, MP\_Lite M-VIA and MVICH provide better performance than TCP. Around 10 KB, both MP\_Lite M-VIA and MVICH switch from the eager protocol to the handshake protocol and start using the RDMA Write mode. There is a little performance decrease at this point, but after 12 KB, the performance increases over TCP.

Figure 4.2 illustrates the matching signature graph of the above message transfer. The signature graph clearly shows the latency, which coincides with the time of the first data point on the graph, of each communication library.

M-VIA based communication libraries provide much lower latency than raw TCP. MP\_Lite M-VIA has the lowest latency at  $40\mu s$ . MVICH and LAM MPI M-VIA are  $45\mu s$  and  $56\mu s$  respectively. Compared to TCP at  $52\mu s$  and MPICH at  $121\mu s$ , M-VIA based libraries have advantages for codes that send many small messages. The M-VIA OS bypass mechanism and

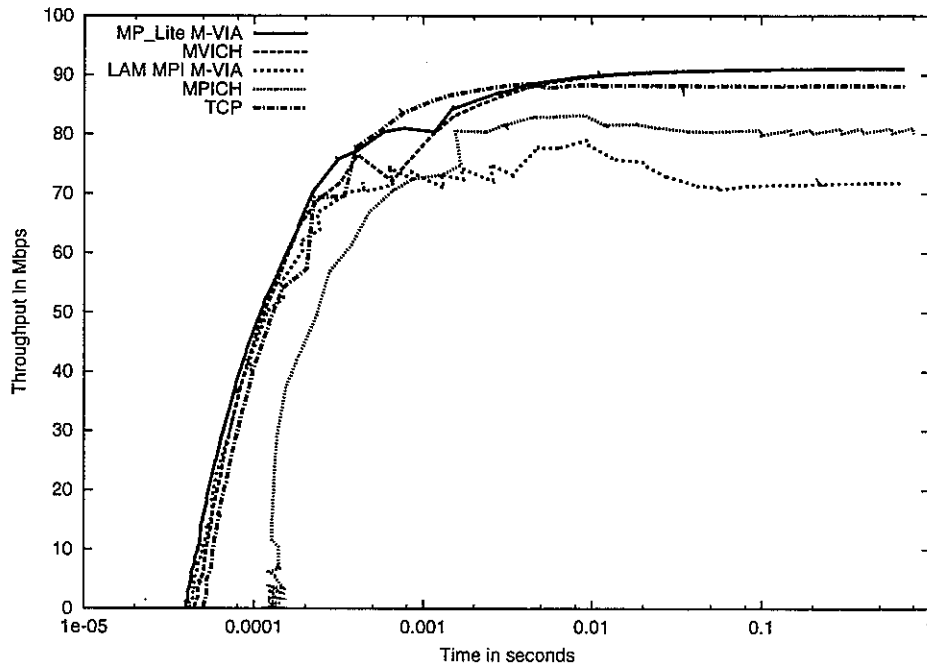


Figure 4.2 The communication latency between Fast Ethernet cards

eager transfer protocol both contribute to the low latency and the characteristics of these libraries.

#### 4.2.2 Gigabit Ethernet on the PC Cluster

The difference between the message-passing libraries is more evident for faster networks such as Gigabit Ethernet. Gigabit Ethernet, also known as the IEEE 802.3z standard, offers a 1 Gbps raw bandwidth which is 10 times faster than Fast Ethernet. It operates in a very efficient full-duplex, point-to-point mode in our experimental configuration. Initially Packet Engine II Hamachi cards were used as our test NICs, but they can deliver at most 330 KB of data due to some bugs in the device driver. Therefore, we switched to Syskonnect Gigabit Ethernet cards.

Figure 4.3 shows that MP\_Lite M-VIA and MVICH reach a maximum of 425 Mbps. Compared to raw TCP, which has a 290 Mbps maximum, the result is very impressive. TCP based MPICH tops out at 230 Mbps, which is only a little more than half of MP\_Lite M-VIA and MVICH. For messages sizes between 2 KB and 16 KB, the throughput of MP\_Lite M-VIA

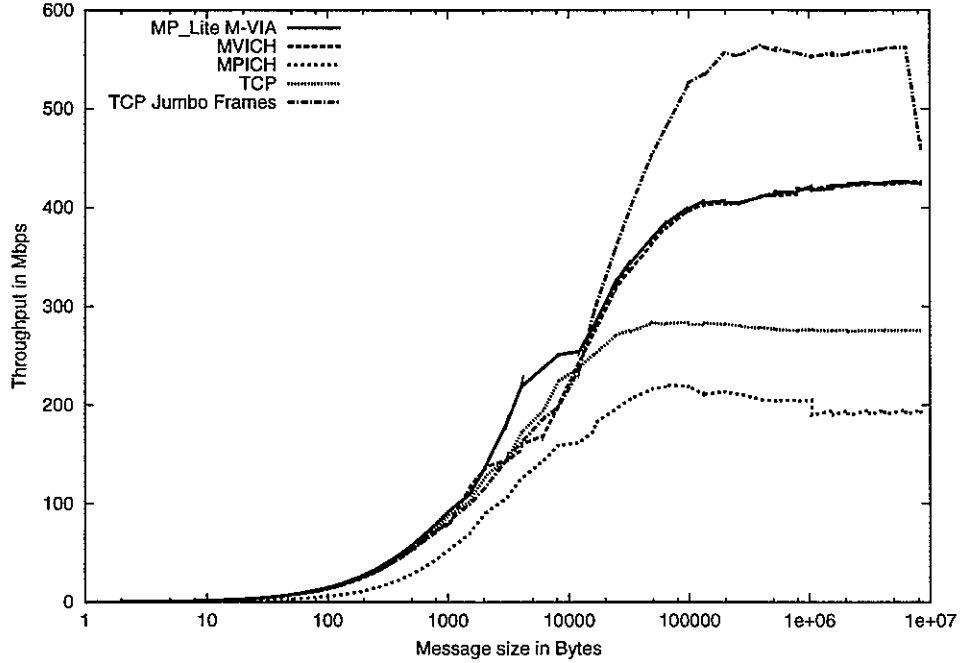


Figure 4.3 The throughput between Syskonnect Gigabit Ethernet cards on the PC test cluster

is much better than MVICH. This is because MP\_Lite can use larger buffers to send small messages without increasing the system memory usage much.

The latency of MP\_Lite M-VIA is  $45\mu s$ , which is the best of the communication libraries tested. The  $51\mu s$  latency of MVICH is also very low. TCP and MPICH are at  $53\mu s$  and  $127\mu s$  respectively.

The Syskonnect Gigabit Ethernet cards support TCP jumbo frames, in which the MTU (Maximum Transfer Unit) of 9000 bytes is used instead of the standard 1500 bytes. Figure 4.3 shows that by enabling jumbo frames, the performance of TCP will reach 580 Mbps. The latency remains the same as with the standard MTU. The native MTU of M-VIA is only 1480 bytes, and currently it does not support jumbo frames. It would be nice to run MP\_Lite M-VIA in conjunction with jumbo frames in the future.

Although the MPICH we tested is based on TCP, enabling jumbo frames does not improve the performance of MPICH. This is because MPICH initializes the TCP buffer to a fixed 4096 bytes, thus a large MTU does not improve the performance of MPICH much (OF00).

### 4.2.3 Gigabit Ethernet on the Alpha Cluster

This section focuses on the performance of the communication on the Alpha Linux cluster connected by Syskonnect Gigabit Ethernet cards. Figure 4.4 illustrates the throughput as a function of message size for MP\_Lite M-VIA, MPICH, TCP and TCP with jumbo frames. The curve for MVICH is not shown here because currently MVICH does not work on Alpha workstations. Figure 4.5 is the corresponding signature graph, which shows the latency (the time of the first data point) of each communication library.

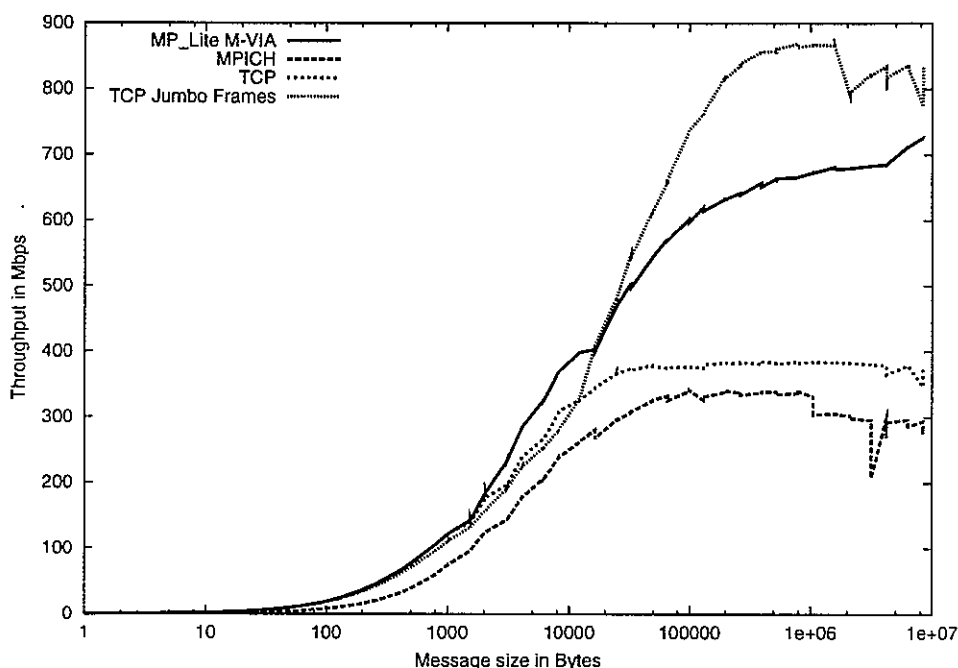


Figure 4.4 The throughput as a function of message size on the Alpha cluster

The performance of each communication library on the Alpha platform is much better than on PCs due to less strain put on the memory bus. The maximum throughput of MP\_Lite M-VIA is as high as 720 Mbps, with a  $36\mu s$  latency. The throughput of raw TCP and MPICH are 390 Mbps and 350 Mbps respectively, with latencies of  $38\mu s$  and  $93\mu s$ .<sup>1</sup>

The TCP with jumbo frames again has the highest 880 Mbps maximum throughput. How-

<sup>1</sup>The results are tested using Linux non-SMP kernel. Using SMP kernel will greatly increase the latency of TCP.

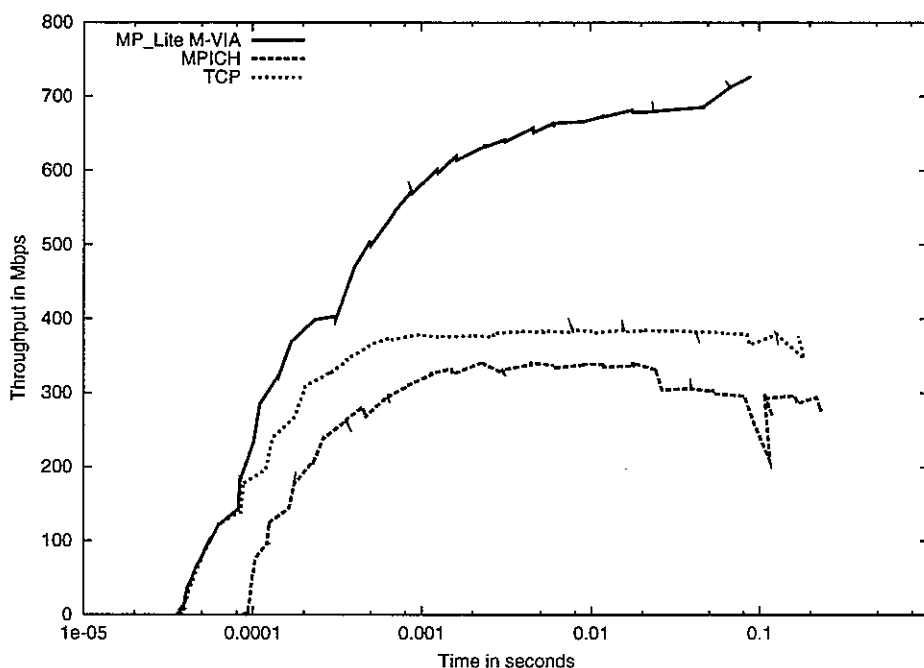


Figure 4.5 The signature graph on the Alpha cluster

ever, this requires a switch that supports jumbo frames, which limits its use currently. The support of jumbo frames in M-VIA is expected in future releases. The performance of MPICH actually decreases by enabling jumbo frames.

### 4.3 Channel-Bonding on Linux Clusters

Channel-bonding is the ability to stripe messages across multiple NICs to increase the communication rate between machines. Figure 4.6 shows that channel-bonding three 3Com Fast Ethernet cards on PCs triples the communication bandwidth. Channel-bonding four Fast Ethernet cards provides 332 Mbps, or nearly 90% of the potential bandwidth. However, the tested M-VIA does not have full reliability built in yet, but these results are encouraging.

Currently we can use three 3Com cards or two Tulip cards for channel-bonding. Using the fourth 3Com card or the third Tulip card can pass the NetPIPE test, but exhibits errors during bi-directional transfers. We are unable to install the fourth Tulip driver on the Linux system, and unable to install two Intel/Pro 100 M-VIA drivers.

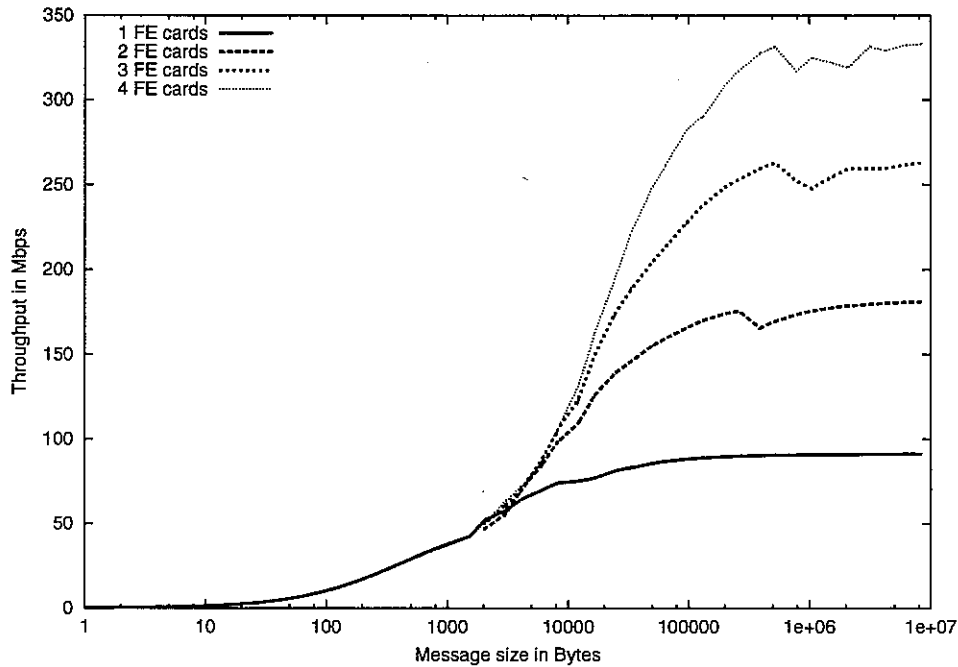


Figure 4.6 Channel-bonding up to four 3Com Fast Ethernet cards between PCs

Figure 4.7 is the result of channel-bonding two Syskonnect Gigabit Ethernet cards on Alpha systems. The result is not as good as on PCs. Using two Gigabit Ethernet cards only offers a 20% improvement, nearly 150 Mbps extra bandwidth over using a single NIC. Because M-VIA still has one memory copy on the receive side, the performance is limited by the internal memory bandwidth, which limits the flow of data through the PCI bus.

#### 4.4 Summary

In this chapter, the performance of MP\_Lite M-VIA, MVICH, MPICH, LAM MPI M-VIA, TCP and TCP with jumbo frames using Fast Ethernet and Gigabit Ethernet cards on both the PC and Alpha platforms are presented. Generally, VIA based communication libraries have better performance on throughput and latency. MP\_Lite M-VIA has impressive performance on both Fast Ethernet and Gigabit Ethernet. It has the lowest latency and nearly double the performance of MPICH. The low latency is achieved by the M-VIA operating system bypass mechanism for reducing system overhead, and by using the eager communication protocol as

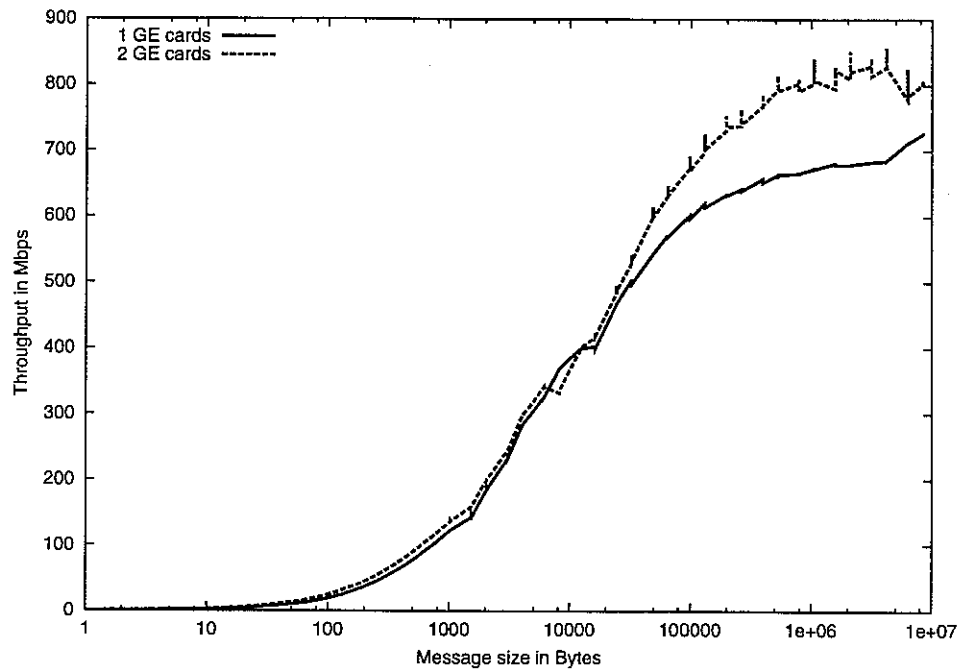


Figure 4.7 Channel-bonding two Gigabit Ethernet cards on the Alpha cluster

well as buffering mechanisms to reduce the transfer delay. The small message header and large buffers also reduce the communication overhead for small messages. The higher throughput is obtained because of the very efficient RDMA Write mechanism. The memory copies are minimized. Although reliability support needs to be further optimized and tested, the results are very promising.

Channel-bonding of three Fast Ethernet cards provides a nearly ideal tripling of the communication rate. This is a good way to increase the communication performance without greatly increasing the overall system cost. Although we can channel-bonding two Gigabit Ethernet cards, the performance improvement is not as much as for Fast Ethernet cards, due to the limitation of the internal memory bandwidth.

## CHAPTER 5. DISCUSSION AND CONCLUSIONS

This chapter will give a summary of the implementation and discuss the limitations and issues of M-VIA and the MP\_Lite library. Possible countermeasures and future work will also be proposed.

### 5.1 Features

The design and implementation of MP\_Lite for M-VIA has achieved several objectives: high performance, channel-bonding capability, portability, and a user friendly system.

#### 5.1.1 High Performance

The high performance of MP\_Lite M-VIA is demonstrated in the low latency and maximum throughput. The implementation also tries to use wait functions instead of polling functions to minimize the CPU load.

For both Fast Ethernet and Gigabit Ethernet, MP\_Lite M-VIA has a much lower latency than MPICH, and is also better than MVICH. The OS-bypass mechanism of M-VIA and the light-weight nature of the MP\_Lite library are the main factors that contribute to the low latency. However, the following implementation choices are also important:

1. The eager protocol sends small messages without delay.
2. Pre-registered buffers are used to send and receive small messages to avoid dynamic memory registration, which is more expensive than memory copies for small messages.
3. The small message envelop (message header) reduces the overhead.



For large messages, the handshake protocol triples the latency. However sending a large message requires much more time, so the latency is not a significant part of the total communication time. The time to send the data essentially hides the extra latency.

MP\_Lite M-VIA has much better throughput compared to MVICH if the message size is smaller than 16 KB. This is because MP\_Lite M-VIA uses larger buffer size so that a small message can be sent in one descriptor. The large buffer reduces the overhead of data segmentation, assembly and transmission. Using large buffer does not increase the system memory usage in MP\_Lite M-VIA. For larger messages, MP\_Lite M-VIA and MVICH have almost the same throughput. Both can deliver almost all the performance that M-VIA provides. The high throughput for large messages is due to the highly efficient RDMA mechanisms that reduce the extra memory-to-memory copies.

### 5.1.2 Channel-Bonding

MP\_Lite M-VIA can safely use at least three network interface controllers simultaneously on a computer to increase the potential bandwidth. Channel-bonding three Fast Ethernet cards triples the maximum throughput without increasing the cost greatly. Using four Fast Ethernet cards has the potential to further increase the the maximum throughput, but this is still under development and testing. MP\_Lite M-VIA is the first channel-bonding implementation on M-VIA. Neither MPICH (MVICH) or LAM MPI have this capability.

### 5.1.3 Portability

MP\_Lite M-VIA is programmed using the API defined by the VIA specification. The implementation does not rely on M-VIA in any way. Therefore, the module should be able to use other VIA-enabled networks without much modification. Furthermore, we have ported the current release of M-VIA to the Alpha architecture running Linux. The performance of MP\_Lite M-VIA is also good on Alpha.

### 5.1.4 User Friendly System

MP\_Lite M-VIA provides the same interface for applications as other MP\_Lite modules. MP\_Lite M-VIA will automatically determine the devices to be used. Except for installing and configuring the M-VIA software package, or another M-VIA network system, no extra configuration work is required to run MP\_Lite M-VIA. The execution procedures and command line arguments are exactly the same as for other MP\_Lite modules. There is also debugging information available if compiled with the requisite debug options.

## 5.2 Limitations

As a research project, the implementation of MP\_Lite on M-VIA exploits the basic functionality and performance potential of M-VIA. Although the results are encouraging, there are still many issues that may further improve the performance.

### 5.2.1 Reliability

The VIA supports three levels of communication reliability at the NIC level: unreliable delivery, reliable delivery and reliable reception. Reliable reception has the highest level of overall reliability, and is necessary before MP\_Lite M-VIA is practical for real applications.

An unreliable delivery VI guarantees that data will arrive on the receiving side at most once and the corrupted data will be detected. The data may be lost, or arrive in an erroneous order. The VI will not re-transmit data when these errors occur.

For the reliable delivery mode, data will arrive at the destination exactly once, and in the order submitted. This requires that the destination side replies with an acknowledgment to the source, either in a stand-alone package or by a piggy-backing mechanism to include the acknowledgment in the next set of data sent.

For reliable reception, in addition to the requirements of reliable delivery, the transmission is successful only when the data has been delivered into the targetted user memory. This level of reliability is not yet supported in the current M-VIA release.

Table 5.1 lists the features of these reliability levels. (CCC97)

Table 5.1 Reliability guarantees

Property	Unreliable	Reliable Delivery	Reliable Reception
corrupt data detected	yes	yes	yes
data delivered at most once	yes	yes	yes
data delivered exactly once	no	yes	yes
data order guaranteed	no	yes	yes
data lost detected	no	yes	yes
connection broken on error	no	yes	yes
state of send/RDMAW when request completed	in-flight	in-flight	completed on remote end also
state of send/RDMAW when error occurs	unknown	unknown	first one unknown, others not delivered

M-VIA version 1.2b2 supports reliable delivery. It introduces the windows and acknowledgments to enhance the transmission reliability between sending and receiving VIs in the generalized Ethernet ring device layer, which is on top of the device driver layer. It is not surprising that the performance is slightly degraded due to the added handshaking and re-transmission. Our experiments show that the latency is increased by  $10\mu s$ , and the throughput has a 5% degradation, when compared to unreliable service.

M-VIA version 1.2b2 does not support reliable reception. The current implementation of MP\_Lite M-VIA associates two sequence numbers for each connection to improve error detection. The *next sequence number* holds the number for the next sending packets. Each data packet is sent alone with the sequence number. The *sequence number expected* field records the next expected packet. If the received sequence number does not coincide with the sequence number expected, data has been lost. Data will not be duplicated because even for unreliable service, data is only delivered once. Sequence numbers provide a simple method to detect data lost in some situations. However, for messages sent by using an RDMA Write, since receiving VI does not consume descriptors except for the last packet, the system is unable to detect a packet lost by using the added sequence number.

Implementing reliable reception may add more overheads and impact performance, but it should be minimal. Most of the time-critical overhead has been added in the reliable delivery service.

### 5.2.2 Resource Reservation

Each receive VI pre-posts some descriptors (buffers) to receive unexpected data. In MP\_Lite M-VIA, the buffer resource is organized as the *mbuf* data structure. Such buffer reservations should be done before the connection is set up. If the descriptors are posted after the connection is in place, and the data arrives before buffers are posted, there will be no place to hold the data, so it will be silently lost. This data loss, due to insufficient pre-posting of buffers, will only happen for small messages sent using the eager protocol. For the RDMA write mode, the data transfer can start only after the destination buffer is ready.

One question is how many buffers should be reserved and the size of each buffer. Suppose that a 32 KB buffer block is associated with each descriptor, which is the maximum, and 10 such descriptors are posted in each VI. The total buffer reserved for each VI is 320 KB. Also assume that the average size of short messages is 8 KB. In this configuration, at most 10 unexpected short messages can be received without posting any receive (each message will consume one descriptor). If a 4 KB block is associated to each descriptor, and the total buffer space reserved is still 320 KB, then 80 descriptors need to be posted. Each short message of size 8 KB will consume 2 descriptors, and 40 unexpected messages can be received, which is more than the first configuration. Although using a smaller buffer block is more efficient for resource utilization, sending messages larger than the block size requires the consumption of more than one descriptor and multiple sends. This impacts the overall performance. In the actual implementation of MP\_Lite M-VIA, we chose the size of 16 KB. This could be tuned, either larger or smaller, for specific applications.

The buffer reservation has scalability issues. In a system that has 64 nodes, on every node, 63 VIs need to be created and 63 connections need to be setup. If a 320 KB buffer is reserved for each VI, the total buffer reserved is at least  $320KB \times 63 = 20MB$  in each node, which is impossible because the maximum memory region that can be registered in the M-VIA implementation is currently 16 MB.

A better solution is to have a flow control mechanism. The source node has an initial window that tells how many buffers are available on the destination. Whenever it sends out a

packet, it decreases the window size by one. It does not send out more packets if the window size becomes zero. The destination node informs the source node of the availability of buffers in a timely manner. By using this technique, the risk of insufficient buffers and the resulting loss of data can be eliminated. Each VI can safely pre-post a small number of buffers. This procedure may have a performance penalty because of the overhead of transmitting window size information. There are complicated optimization techniques available such as Silly Window Syndrome (Com95).

### 5.2.3 Channel-Bonding Issues

Channel-bonding provides higher bandwidth, but requires more memory and marginally increases latency. Each network interface needs a copy of the related data structures. The resources reserved as discussed in the previous subsection will also be doubled if using two interfaces. The connection set up procedure will be impeded because more connections are to be established. This may also lead to scalability problems.

### 5.2.4 Overlapping Communication and Computation

Overlapping communication and computation is a nice way to improve the performance of parallel applications, if they can adequately take advantage of it. This requires the use of non-blocking asynchronous communications. An application posts a send or receive to start the communication, then continues working on the computation. The communication and computation are performed concurrently until the application calls the wait function to finish the communication. Overlapping can give a speedup of at most a factor of 2.

One thing related to the performance of overlapping is the processor overhead in the communication subsystem or what is left over for the application. A polling implementation usually leads to a heavy CPU workload, and therefore leaves little for the application to use during overlapped communication and computation.

MP\_Lite M-VIA currently does not support overlapping communication and computation. For non-blocking `MP_ASend()` and `MP_ARecv()` functions, we just put the message into the

send\_q or recv\_q. It is MP\_Wait() that actually performs the communication. The reason for this is that we are using the M-VIA blocking send and receive functions, which wait on the VI send and receive queues. So the communication can not be started before the MP\_Wait() function call.

The handshake protocol also limits the ability to overlap communication and computation. The source node needs to wait for a reply after sending out the request, and the receiver is required to wait for the request before replying with the destination buffer address.

One solution is to use the M-VIA asynchronous communication. M-VIA does provide some asynchronous communication functions. They are implemented as signal notification mechanisms. Whenever a send or receive descriptor has completed, a call-back function is called to notify the completion. However, these asynchronous functions have not been fully optimized yet. In the current version of M-VIA, the asynchronous receive takes three times the latency compared to the blocking function. Asynchronous communications are quite promising and need to be explored in the future.

Another way to overlap communication and computations is to use threads. A communication thread can be created to control the transfer of the messages. The communication thread works concurrently with the main thread. Either blocking or non-blocking communications can be implemented with the communication thread. A synchronization method, such as semaphores or mutexes, would be required to synchronize the thread interactions. This would solve the contention between the main thread and the communication thread.

The disadvantage of the thread based approach would be the synchronization delay. The scheduling of threads would add latency to the communication. The current M-VIA VIPL (VI Provider Library), is also not a thread safe library. Explicit locking is required when multiple threads are accessing the same queue within a VI.

### 5.2.5 Other Issues

The number of VI connections is one of the scalability issues. MP\_Lite M-VIA requires a fully connected network. Large configurations will introduce significant delay in the con-

nection start up procedure. A simple solution is to establish connections only when they are needed. Some applications do not require a fully connected network. For example, applications using a tree-like structure for communications may only need to establish connections between different tree layers. For these applications, establishing connections only when send or receive operations are requested may reduce the initialization overhead. However, this has several drawbacks: performance degradation that each communication operation may incur connection setup and breakdown overheads. Only problem is when the connection needs to be established. Since the connection setup procedure requires the co-operation of the source and the destination nodes, if the send and receive pairs are not the exact match, some complicated buffering and re-connect mechanisms may be required. Moreover, if a wildcard (MPI\_ANY\_SOURCE) is used in a receive function, a fully connected network may be needed. One possible solution is to assign a “master” node. Each node establishes a connection to the master node initially. The connection request to other nodes can via the master node. However, this will increase the work load of the master node.

Another issue is the dynamic memory registration. The use of DMA to transfer data directly into and out of user buffer requires that the data page be locked and cannot be paged-out by the operating system. To avoid an extra memory copy, user buffers need to be dynamically registered before data transfer and deregistered when the transfer is completed. Currently we only have a simple memory registration cache to keep the last few sets of registration information. Without a more efficient memory registration manager, the frequent registration and deregistration of large buffers may be too expensive, and lead to fragmentation of the page tables (SASB99; BM00).

### 5.3 Conclusions and Future Efforts

The implementation of MP\_Lite for M-VIA incorporates the efficiency of MP\_Lite with the high performance features of M-VIA, resulting in a small, high-performance message-passing library that has much lower latency and better throughput on both Fast Ethernet and Gigabit Ethernet. The eager protocol and the handshake protocol provide a better balance between

latency and throughput for different message sizes. Channel-bonding based on the VIA is a unique feature of MP\_Lite M-VIA, providing from double to triple the performance of a single network interface.

The limitations discussed in the previous section imply that further improvement is possible in a number of directions:

- Improved reliability
- Asynchronous or thread-based communications
- More testing
- Application utilization

The VIA is supposed to work on System Area Networks, which are usually connected by fabrics that have very low error rates. For networks such as traditional LANs, it is important to provide full reliability support for the upper layers. As discussed in the previous section, M-VIA currently does not support reliable reception. This limits its overall applicability. Because reliable reception and reliable delivery are very similar, it is expected that the implementation will not degrade performance much. The MP\_Lite M-VIA module does not need any modification to support higher reliability because it will automatically choose the highest reliability level supported by the underlying network interface controller.

A flow control mechanism in MP\_Lite M-VIA would be useful. The current MP\_Lite M-VIA assumes the data destination has pre-posted enough buffers to receive unexpected small messages, which is the usual case. However, if an application continuously sends many small messages without posting any receives, the destination may run out of buffer resources. Currently only error messages will be generated in this situation. Using data windows to control data flow as discussed in the previous subsection is a better solution.

It may be beneficial to improve the asynchronous communication so that communication and computation can be overlapped. Asynchronous communication (signal-based) or the use of threads are two approaches. They need careful design and implementation so that perfor-



mance will not be overtly impacted. A possible method is to combine them with synchronous communications.

More testing is needed for MP\_Lite M-VIA to improve the stability and usability. Currently it is quite stable for running on small clusters and can successfully run some benchmarks and real applications such as the Ames Lab Classic Molecular Dynamics program. Further testing is still required, for more applications and larger configurations. We are currently building a channel-bonded PC clusters with 24 nodes and three 3Com cards per machine. It is also imperative that we test the functionality on other VI-enabled networks, such as Giganet.

## BIBLIOGRAPHY

- [aUoP] Computer Engineering Groups at University of Parma. Parma2: porting VIA in LAM/MPI. <http://ce.unipr.it/research/parma2/via/via.html>.
- [BBCR] Massimo Bertozzi, Franco Boselli, Gianni Conte, and Monica Reggiani. An MPI implementation on the top of the Virtual Interface Architecture. *PVM/MPI*, pages 199–206.
- [BDV94] Greg Burns, Raja Daoud, and James Vaigl. LAM: An open cluster environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.
- [BGC98] Philip Buonadonna, Andrew Gaweke, and David Culler. An implementation and analysis of the Virtual Interface Architecture. In *Proceedings of SC98*, pages 7–13, 1998.
- [BM00] Ron Brightwell and Arthur B. Maccabe. Scalability limitations of VIA-based technologies in supporting MPI. In *Proceedings of the Fourth MPI Developer's and User's Conference*, March 2000.
- [BS99] Patrick Bozeman and Bill Saphir. A modular high performance implementation of the Virtual Interface Architecture. *USENIX Annual Technical Conference*, 1999.
- [CCC97] Compaq Computer Corp., Intel Corporation, and Microsoft Corporation. Virtual Interface Architecture specification. <http://www.viarch.org/>, 1997.
- [Cen] National Energy Research Scientific Computing Center. MVICH - MPI for Virtual Interface Architecture. <http://www.nersc.gov/research/ftg/mvich/index.html>.

- [CJRS89] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.
- [CKS<sup>+</sup>00] Weiyi Chen, Ricky A. Kendall, Ron L. Shepard, Mike Minkoff, and Mike Dvorak. A generic compression library for high performance distributed application. Scalable Computing Laboratory, Ames Laboratory, 2000.
- [Com95] Douglas E. Comer. *Internetworking with TCP/IP Vol I: Principles, Protocols, and Architecture, The 3rd Edition*. Prentice Hall, Inc., Upper Saddle River, NJ, 1995.
- [CS99] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1999.
- [DBL<sup>+</sup>97] Cezary Dubnicki, Angelos Bilas, Kai Li, , and James Philbin. Design and implementation of virtual memory-mapped communication on myrinet. In *Proceedings of the International Parallel Processing Symposium*, pages 388–396. Apr. 1997.
- [Dem95] James Demmel. Lecture notes for introduction to parallel computing. <http://www.cs.berkeley.edu/~demm/lect/cs267>, 1995.
- [DRM<sup>+</sup>98] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert. Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The Virtual Interface Architecture. *IEEE Micro*, pages 66–76, 1998.
- [DS98] Rossen Dimitrov and Anthony Skjellum. Efficient MPI for Virtual Interface (VI) Architecture. <http://www.mpi-softtech.com/publications/empivia.ps>, 1998.
- [DS99] Rossen Dimitrov and Anthony Skjellum. An efficient mpi implementation for virtual interface (vi) architecture-enabled cluster computing. In *Proceedings of the MPI Developers Conference*, pages 15–24, 1999.

- [fIT00] American National Standard for Information Technology. Scheduled Transfer Protocol (ST). <http://www.hippi.org/cST.html>, 2000.
- [For94] The MPI Forum. <http://www.mpi-forum.org>, 1994.
- [For95] The MPI Forum. MPI: A message-passing interface standard. <http://www-unix.mcs.anl.gov/mpi/index.html>, 1995.
- [For97] The MPI Forum. MPI-2: Extensions to the message-passing interface. <http://www-unix.mcs.anl.gov/mpi/index.html>, 1997.
- [GLnDS96] William Gropp, Ewing Lusk, nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.
- [NHL96] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.
- [OF00] Hong Ong and Paul A. Farrell. Performance comparison of LAM/MPI, MPICH, and MVICH on a linux cluster connected by a gigabit ethernet network. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, Georgia, 2000.
- [Pac97] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers. Inc., San Francisco, CA, 1997.
- [Pan] Avneesh Pant. A high performance MPI implementation on the NTSC VIA cluster. <http://archive.ncsa.uiuc.edu//General/CC/ntcluster/VIA/MPI-FM-VIA/fm-via.htm>.
- [PKC97] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast messages (FM): Efficient, portable communication for workstation clusters and massively-parallel processors. *IEEE Concurrency*, 5(2):60–72, April-June 1997.

- [PT98] L. Prylli and B. Tourancheau. BIP: a new protocol designed for high performance networking on myrinet. Workshop PC-NOW, IPPS/SPDP98, 1998.
- [PVM] The PVM system. <http://www.netlib.org/pvm3/book/node17.html>.
- [SASB99] Evan Speight, Hazim Abdel-Shafi, and John K. Bennett. Realizing the performance potential of the virtual interface architecture. In *International Conference on Supercomputing*, pages 184–192, 1999.
- [SGI] SGI. Scheduled Transfer Protocol on Linux. <http://oss.sgi.com/projects/stp/>.
- [SHM94] SHMEM technical note for C. Technical Report SG-2516 2.3, Cray Research, Inc., October 1994.
- [SMG97] Quinn O. Snell, Armin R. Mikler, and John L. Gustafson. Netpipe: A network protocol independent performance evaluator. Scalable Computing Lab/Ames Laboratory, Ames, IA, 1997.
- [SSB<sup>+</sup>95] T. Sterling, D. Savarese, D. Becker, B. Fryxell, and K. Olson. Communication overhead for space science applications on the beowulf parallel workstation. In *Proceedings of the Fourth IEEE Symposium on High Performance Distributed Computing (HPDC)*, pages 23–30, 1995.
- [Sun90] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.
- [TCK01] Dave Turner, Weiyi Chen, and Ricky Kendall. Performance of the MP\_Lite message-passing library on Linux clusters. In *Linux Clusters: HPC Revolution*, 2001.
- [Tur] David Turner. MP\_Lite: A light weight message passing library. [http://cmp.ameslab.gov/MP\\_Lite/](http://cmp.ameslab.gov/MP_Lite/).
- [vEBBV95] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings*

*of the 15th ACM Symposium of Operating Systems Principles*, volume 29, pages 40–53, Dec 1995.

- [vECgS92] Thorsten von Eicken, David E. Culler, Seth Copen goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266. ACM Press, May 1992.