© Copyright by Gregory Allen Koenig, 2003

AN EFFICIENT IMPLEMENTATION OF CHARM++ ON VIRTUAL MACHINE INTERFACE

BY

GREGORY ALLEN KOENIG

B.S., Indiana University-Purdue University Fort Wayne, 1993
B.S., Indiana University-Purdue University Fort Wayne, 1995
B.S., Indiana University-Purdue University Fort Wayne, 1996

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2003

Urbana, Illinois

Abstract

Charm++ is a message-driven parallel programming language designed with the goal of enhancing programmer productivity by providing a high-level abstraction of a parallel computation while at the same time providing good performance on platforms ranging from traditional supercomputers to more recent commodity cluster environments. Charm++ is based on the C++ programming language and is backed by an adaptive runtime system that provides features such as processor virtualization, load balancing, and communication optimizations. Programs written in Charm++ consist of parallel objects called *chares* that communicate with each other through asynchronous message passing. When a chare receives a message, the message triggers a corresponding method within the chare object to handle the message asynchronously. Charm++ is implemented on top of a software layer called Converse which supports portability across multiple platforms and, in particular, provides interprocess communication.

Virtual Machine Interface (VMI) is a high-bandwidth low-latency communication layer designed with the primary goal of providing a single programming interface to the various system area networks commonly used in modern commodity clusters. VMI is generally not intended for direct use by application developers but rather to provide a low-overhead communication layer to developers of higher level programming languages and message passing libraries. A language or library implemented on VMI immediately gains access to all of the network interconnects supported by VMI while paying a small overhead of only a few microseconds per message. Furthermore, the language or library can take advantage of other VMI features such as the ability to stripe data across multiple network interfaces, auto-

matic fail-over from one network transport to another, access to communication transports for distributed grid-based computing, and the ability to monitor and dynamically tune the communication layer.

This thesis describes an efficient implementation of Charm++ on Virtual Machine Interface and discusses the various design trade-offs involved. Performance of the implementation is evaluated for latency and bandwidth and compared to the performance of Charm++ implementations running on other communication layers.

To my parents, who taught me that the most important things in life cannot be learned from a book.

Acknowledgments

I would like to thank my adviser, Professor Laxmikant Kalé, who gave me a huge degree of independence in working on this project. His patience and willingness to allow me to explore various facets of the problem certainly contributed greatly to the value that I received from doing this thesis and transformed the project from a mere item on an academic checklist into a true learning experience.

I am also indebted to my colleagues in the Advanced Cluster Group at the National Center for Supercomputing Applications. For the past several years they have not only provided an environment of technical excellence that has challenged me intellectually but they have also been good friends. I specifically thank Avneesh Pant who has taught me an immense amount about writing efficient code and has given me guidance on many of the difficult details in this thesis. I also thank Rob Pennington who originally encouraged me to seek employment with NCSA and who has helped me navigate the terrain of the scientific community several times.

Although I have only recently joined their ranks, I would like to thank the members of the Parallel Programming Laboratory in the Department of Computer Science at the University of Illinois at Urbana-Champaign for making me feel immediately welcome in the group and for giving me the benefit of their years of experience with the Charm++ and Converse systems. Among them, Orion Sky Lawlor answered countless questions in the early stages of this project and saved me many hours of examining source code.

Finally, I give special thanks to my parents and sister who have always encouraged me to pursue my goals in life.

Table of Contents

List of Tables . . . ix

List of Figures . . . x

Chapter 1 Introduction . . . 1
1.1 Charm++ . . . 2
1.2 Virtual Machine Interface . . . 3
1.3 Thesis Contributions . . . 4
1.4 Thesis Organization . . . 5

Chapter 2 Related Work . . . 6
2.1 Internet Protocol . . . 6
2.2 Myrinet . . . 7
2.3 InfiniBand . . . 8
2.4 Message Passing Interface . . . 9
2.5 Virtual Machine Interface 1.0 . . . 10
2.6 Globus . . . 11

Chapter 3 Charm++ . . . 13
3.1 Chares . . . 15
3.2 Message-Driven Execution . . . 15
3.3 Converse . . . 16
3.4 Charm++ Program Files . . . 17
3.5 Additional Features . . . 20

Chapter 4 Virtual Machine Interface . . . 23
4.1 VMI Network Stack . . . 25
4.2 I/O Request Blocks . . . 26
4.3 Message Streams . . . 29
4.4 Remote DMA . . . 32
4.5 Loadable Devices . . . 34
4.6 Monitoring and Management Framework . . . 35

Chapter 5 Implementation Details 37
5.1 Program Startup 37
5.2 Message Sends 39
5.3 Message Receives 44
5.4 Memory Management 45

Chapter 6 Performance 47
6.1 Test Environment 47
6.2 Gigabit Ethernet Performance 49
6.3 Myrinet Performance 50
6.4 Interpretation of Results 51

Chapter 7 Conclusion and Future Work 54

References 57

List of Tables

4.1 IRB commands and their meanings 29
4.2 VMI supported devices 34
4.3 VMI experimental devices 35

List of Figures

3.1 Structure of a parallel application that uses Charm++ 18
3.2 The "Hello World" program .ci file 19
3.3 The "Hello World" program .h file 20
3.4 The "Hello World" program .C file 21

4.1 Structure of an application that uses VMI 27
4.2 Structure of an I/O Request Block (IRB) 28
4.3 Structure of VMI Stream, Slab, Buffer Op, and Buffer [19] 31

5.1 Structure of the efficient implementation of Charm++ on VMI 38

6.1 Performance of Converse on Gigabit Ethernet 49
6.2 Performance of Charm++ on Gigabit Ethernet 51
6.3 Performance of Converse on Myrinet 52
6.4 Performance of Charm++ on Myrinet 52

Chapter 1

Introduction

High-performance parallel computing has emerged as a major field of computer science study that attempts to coordinate the computational resources of multiple processors with the goal of achieving fast execution times for computationally-intensive problems. Parallel computing is interesting because it allows researchers to investigate problems that would otherwise be infeasible, both in terms of the computational time required and in terms of the size of the overall computation.

The primary difficulty with parallel computing is coordinating the efforts of multiple processors in an efficient manner. The problem to be solved must be divided into pieces that can be solved simultaneously and then these pieces must be assigned to processors in the system. Generally, during the course of the computation, the processors in the system must communicate with each other to exchange intermediate results. Writing correct code to carry out this communication can be quite difficult and tedious. Forcing the application programmer to write code to coordinate communication requires the programmer to think at two levels of abstraction: at the lower level of the communication routines, and at the higher level of the domain-specific problem.

Another difficulty with parallel computing is dealing with rapidly-changing technology. By definition, the users of parallel computers are on the leading edge of technology. This technology tends to change at a fast pace, and it is not uncommon for high-performance computers to become obsolete within only a few years. Because scientific applications often

tend to take months or years to write, programmers are reluctant to invest in technology that may not be available for the foreseeable future.

The solution to these difficulties is to provide abstractions to the programmer that make developing parallel applications simpler. Typically, these abstractions are in the form of parallel languages and libraries that encapsulate details of the underlying technology and present a higher-level view of the parallel computation to the programmer. Because the programmer is given a more abstract view of the computation, more attention can be spent on the domain-specific problem and less on the underlying communication involved. Furthermore, when the underlying technology changes, the parallel languages and libraries can simply be redeployed on new technology without requiring changes to the application software.

The fundamental challenge to using abstraction to solve the problems described above is to make the abstractions low-overhead and the associated cost of using them negligible. This is the focus of much of the current research in high-performance parallel computing.

1.1 Charm++

Charm++ [16] is a message-driven parallel programming language designed with the goal of enhancing programmer productivity by providing a high-level abstraction of a parallel computation while at the same time providing good performance on platforms ranging from traditional supercomputers to more recent commodity cluster environments. Charm++ is based on the C++ programming language and is backed by an adaptive runtime system that provides features such as processor virtualization, load balancing, and communication optimizations. Programs written in Charm++ consist of parallel objects called *chares* that communicate with each other through asynchronous message passing. When a chare receives a message, the message triggers a corresponding method within the chare object to handle the message asynchronously. Charm++ is implemented on top of a software layer called Converse which supports portability across multiple platforms and, in particular, provides

interprocess communication.

Charm++ presents a powerful abstraction to application developers. The programmer simply invokes methods on objects, which may exist on the local processor or on a remote processor, and Charm++ handles the details of the communication required for remote method invocation. Furthermore, because the Charm++ system hides the details of the underlying processors and communication from the programmer, the system is free to make dynamic runtime optimizations to the computation by migrating processes from heavily-loaded processors onto more lightly-loaded processors.

Charm++ is described in more detail in Chapter 3.

1.2 Virtual Machine Interface

Virtual Machine Interface (VMI) [21, 19] is a high-bandwidth low-latency communication layer designed with the primary goal of providing a single programming interface to the various system area networks commonly used in modern commodity clusters. VMI is generally not intended for direct use by application developers but rather to provide a low-overhead communication layer to developers of higher level programming languages and message passing libraries. A language or library implemented on VMI immediately gains access to all of the network interconnects supported by VMI while paying a small overhead of only a few microseconds per message. Furthermore, the language or library can take advantage of other VMI features such as the ability to stripe data across multiple network interfaces, automatic fail-over from one network transport to another, access to communication transports for distributed grid-based computing, and the ability to monitor and dynamically tune the communication layer.

By designing a parallel language or library as a layer on top of VMI instead of directly on the underlying communication layers, the language or library is easily deployed on new communication layers as technology changes. A VMI device driver for the new layer is simply

written and the language or library is redeployed with no changes necessary.

Virtual Machine Interface is described in more detail in Chapter 4.

1.3 Thesis Contributions

This thesis describes an efficient implementation of Charm++ on Virtual Machine Interface and discusses the various design trade-offs involved. Performance of the implementation is evaluated for latency and bandwidth and compared to the performance of Charm++ implementations running on other communication layers.

This thesis makes several contributions. First, by implementing Charm++ on VMI, any program written in the Charm++ language immediately gains access to all of the network transports supported by VMI. Currently these transports include TCP/IP [25], VIA [4], Myrinet [1], and InfiniBand [22]. While Charm++ already has support for some of these transports, such as TCP/IP and Myrinet, the support that VMI provides for the these transports appears to offer competitive performance while at the same time providing support for transports such as InfiniBand that Charm++ does not support directly.

Second, the thesis offers contributions in the area of novel features provided by VMI that programs written in Charm++ can now take advantage of automatically. These features include the ability to stripe data across multiple network interfaces, automatic fail-over from one network transport to another, access to communication transports for distributed grid-based computing, and the ability to monitor and dynamically tune the communication layer. Furthermore, as new features are added to VMI, these features are automatically available to Charm++ programs without any modifications.

Due to the previous two contributions, Charm++ programs can see an improvement in performance with little or no effort required on the part of the programmer other than rebuilding the application with VMI support. This is perhaps the most important outcome of this work.

Next, the thesis contributes a mechanism by which the developer can easily examine the structure of communication operations performed by a Charm++ program. By using the VMI monitoring facilities, the developer can match VMI communication protocols with patterns of usage of the higher level Converse and Charm++ libraries. Such capabilities might be useful in the context of distributed grid-based computations in which some communication takes place over high-performance system area networks (i.e., the intra-cluster communication) and other communication takes place over lower-performance local area networks or wide area networks (i.e., the inter-cluster communication).

Finally, this thesis offers contributions in the area of estimating the relative performance of one Charm++ messaging layer to another. For example, by comparing the performance of Charm++ running on VMI with a TCP/IP transport device to the performance of Charm++ running directly on TCP/IP, one may gain insight about the efficiency of the TCP/IP implementation of Charm++.

1.4 Thesis Organization

This thesis contains seven chapters. Chapter 2 presents a survey of work related to this thesis. Chapter 3 provides an overview of the Charm++ programming language and the underlying Converse asynchronous messaging layer while Chapter 4 discusses the details of Virtual Machine Interface. Details of the efficient implementation of Charm++ on Virtual Machine Interface are given in Chapter 5. Chapter 6 presents performance measurements of the implementation for latency and bandwidth benchmarks and compares the performance to that of Charm++ implementations on other communication layers. Finally, Chapter 7 gives concluding remarks and direction for future investigation.

Chapter 2

Related Work

Several projects exist which are related to the technology described in this thesis, either in terms of providing building blocks to this thesis or in terms of presenting similar technologies. This chapter describes such related technologies.

2.1 Internet Protocol

Internet Protocol (IP) [24] along with User Datagram Protocol (UDP/IP) [23] and Transmission Control Protocol (TCP/IP) [25] represent a well-known and robust family of network protocols used for over thirty years in developing both local area network and wide area network applications. In the context of high-performance commodity cluster environments, these protocols are especially important because they are often deployed on top of Ethernet and Gigabit Ethernet [13] network mediums that are often used for both intra-cluster and inter-cluster communication.

The Charm++ net-linux version provides direct support for UDP/IP and the net-linux-tcp version provides direct support for TCP/IP. Generally, the net-linux version is more optimized and offers more favorable performance. Finally, Virtual Machine Interface includes a device driver for TCP/IP. The performance of these three versions of Charm++ are benchmarked and compared in this thesis.

2.2 Myrinet

Myrinet [1] is a local area network technology created by Myricom, Inc. and based on the packet switching network communication used in massively-parallel computers. Through the use of custom VLSI chips, efficient routing techniques, and carefully-designed network control software, Myrinet provides system area network technology capable of supporting hundreds or thousands of nodes.

Myrinet includes several novel features:

- Zero-copy receives – When a Myrinet network adapter receives a packet destined for a process on the local node, it uses DMA to deliver the message data directly into the process's address space without incurring the cost of a transition from user to kernel mode on the processor. This allows the message to be delivered without first copying the message in a temporary kernel-mode buffer as is necessary in traditional messaging layers.

- High data rates – Each connection in a Myrinet system area network is a full-duplex pair of 640 Megabit/second channels.

- Regular topologies and scalability – In contrast to traditional local area network technologies such as Ethernet, Myrinet system area networks are typically arranged in mathematically regular topologies such as hypercubes, two-dimensional meshes, or trees. The aggregate capacity of the network grows with the number of nodes because several packets may be in transit concurrently along different paths.

- Very low error rate – Because Myrinet is designed for more self-contained system area networks instead of larger local area networks, the designers made the assumption that errors would be very rare. This assumption means that minimal overhead is spent on error checking and as a result more of the raw bandwidth of the network is available for message passing.

- Efficient routing – Myrinet uses cut-through routing with flow control on every link. This means that packets are advanced into the outgoing network channel as soon as the header is received and decoded rather than storing the entire packet on each intermediate node.

- Remote DMA put – Myrinet supports RDMA put operations, allowing a sending process to directly place message data into a receiving process's address space. When using RDMA to send a message from a sender to a receiver, the receiver first publishes a buffer in its address space and then the sender writes the message data directly into the buffer. This results in extremely low-latency data transfers for high performance clustering environments.

Applications programmers interact with Myrinet through a message-based communication system called GM [14]. The goals of GM include low CPU overhead, portability, low latency, and high bandwidth.

The Charm++ net-linux-gm version provides direct support for Myrinet. Furthermore, Virtual Machine Interface includes a device driver for Myrinet and therefore Myrinet is available to the efficient implementation of Charm++ on VMI described in this thesis. The performance of these two versions of Charm++ are benchmarked and compared in this thesis.

2.3 InfiniBand

InfiniBand [22] is another network technology similar to Myrinet, based on an open standard that is implemented by several vendors. InfiniBand is designed for use in system area network environments as an interconnect for nodes in high-performance computational clusters and for use in storage area networks providing a low-latency I/O fabric to disk storage.

InfiniBand shares many similar characteristics to Myrinet such as zero-copy receives of message data directly into a process's address space, regular topologies and scalability, very

low error rates, efficient routing, and Remote DMA put operations. Beyond these features, InfiniBand includes some additional features:

- High data rates – InfiniBand hardware exists in three variations in increasing bandwidth. Infiniband 1x provides bandwidths of 2.5 Gigabits/second. Infiniband 4x provides bandwidths of 10 Gigabits/second. Finally, InfiniBand 12x provides bandwidths of 30 Gigabits/second.

- Remote DMA get – In addition to RDMA put operations, InfiniBand also supports RDMA get operations. These operations allow a process to directly read message data directly out of another process's address space without intervention on the part of the remote process.

Virtual Machine Interface includes a device driver for InfiniBand and therefore InfiniBand is available to the efficient implementation of Charm++ on VMI described in this thesis.

2.4 Message Passing Interface

Message Passing Interface (MPI) [7] is an open standard for parallel computing. MPI defines primitives that allow a process to join and leave a computation, determine the size of a computation and the process's rank in the computation, and exchange messages with other processes in the computation. Messages can be sent to a single process, to a user-defined group of processes, or to all processes in a computation. Furthermore, all message send and receive primitives are defined with both synchronous and asynchronous versions.

Charm++ and MPI are similar in that they both provide a more abstract view of a parallel application to the developer. In both systems, the programmer thinks in terms of an ordered set of processes that comprise a computation and communicate with each other via messages. The systems differ in that MPI provides both synchronous and asynchronous

message passing primitives while Charm++ is inherently asynchronous [1]. More importantly, however, Charm++ provides a much more abstract view of a computation than MPI since messages are typically created automatically by the Charm++ system as a result of an invocation of a method on a remote object. That is, the programmer thinks in terms of invoking a method on a remote object instead of thinking in terms of creating and sending messages. In contrast, an MPI programmer thinks specifically in terms of the messages that are sent and received by each process in the computation.

Several implementations of the MPI standard have been created. The two most notable implementations are MPICH [12] and LAM [2].

A version of Charm++ exists which uses MPI as a means of passing messages between processes. This mpi-linux version of Charm++ is commonly used on high-performance supercomputing platforms where the vendor has already provided an efficient MPI layer, thus allowing Charm++ to be deployed on many platforms quickly and easily.

An implementation of MPI has been deployed on VMI. This implementation is based on MPICH 1.2 and will be available as part of the MPICH software distribution in upcoming releases. Furthermore, by using the implementation of MPI on VMI, VMI is available indirectly to Charm++ applications. The efficient implementation of Charm++ on Virtual Machine Interface described in this thesis is implemented directly on top of VMI and thus eliminates the extra MPI layer.

2.5 Virtual Machine Interface 1.0

The efficient implementation of Charm++ on Virtual Machine Interface described in this thesis is based on VMI version 2.0. The predecessor of this version, VMI version 1.0 [21], was designed to enable binary portability of MPI applications across the various clusters at

[1] An asynchronous system can, of course, be extended to provide synchronous operations simply by having the sender of an asynchronous message wait until the receiver responds with an asynchronous acknowledgment message.

the National Center for Supercomputing Applications by allowing applications to run atop various network interconnects without being recompiled. VMI 1.0 defines a basic network abstraction for point-to-point communication and uses dynamically loadable modules to support each network interconnect. Each dynamically loadable module implements a simple API that includes primitives for *connect*, *disconnect*, *send*, and *receive* in terms of whatever a particular network interface provides.

VMI 2.0 [19] is a complete rewrite of the Virtual Machine Interface and improves on the design of VMI 1.0 by providing features such as the ability to stripe data across multiple network interfaces, automatic fail-over from one network transport to another, access to communication transports for distributed grid-based computing, and the ability to monitor and dynamically tune the communication layer.

An implementation of MPI exists for VMI 1.0 and thus VMI 1.0 is available indirectly to Charm++ applications through an mpi-linux-vmi version. The efficient implementation of Charm++ on Virtual Machine Interface described in this thesis is implemented directly on top of VMI and thus eliminates the extra MPI layer.

2.6 Globus

Globus [8] is a programming toolkit that allows a developer to easily create distributed *grid computing* [9] applications. Using Globus, a programmer can construct applications that utilize computational resources at multiple geographically-distributed locations. For example, a Globus application could collect data from scientific devices at two different sites, perform a computation with the data on a supercomputer at a third site, and display the results of the computation with visualization tools and equipment at a fourth site. While it is possible to create such applications without the use of Globus, the goal of the Globus project is to provide abstractions that give a programmer a very high-level view of such distributed resources and allow the programmer to treat them as a single resource called a

metacomputer, thus greatly simplifying the process of developing applications that use these resources.

The Globus toolkit contains components to facilitate communication, authentication, network information collection, and remote data access. Of these, the communication components are probably the most similar to Charm++ technologies. One communication component is the Nexus communication library [11, 10]. When a message is received by a Nexus process, the message automatically triggers a handler function in the process's address space to deal with the message asynchronously. This mechanism allows for the overlapping of waiting for messages with the execution of ready processes. In this way, the Nexus communication library is very similar to the Converse library used by Charm++.

The communication aspects of Globus are also similar to some of the grid computing capabilities available in VMI 2.0. Globus includes a grid-aware implementation of MPI called MPICH-G2 [17]. This version of MPI uses Globus services to coordinated processes on distributed resources to create a single MPI computation. Similarly, VMI includes an implementation of MPI that has the capability of coordinating distributed processes into a single computation.

Chapter 3

Charm++

Charm++ [16] is a parallel object-oriented language based on C++ [26]. The goal of Charm++ is to reduce the complexity of developing parallel programs by addressing a few key issues:

- Portability – The field of high-performance computing changes rapidly. It is not uncommon for parallel computers to become obsolete within only a few years. Because scientific applications often tend to take months or years to write, programmers are reluctant to invest in technology that may not be available for the foreseeable future. Charm++ addresses this problem by providing a portable platform that programmers can use to develop applications that can run unchanged on a wide variety of parallel machines.

- Modularity – Modularity and code reuse are well understood concepts in sequential software. Because parallel programs are inherently complex, modularity is even more important in the realm of high-performance computing. Charm++ provides mechanisms for allowing parallel algorithms to be encapsulated within reusable modules that can be easily leveraged in multiple programs.

- Latency tolerance – In most parallel computers, accessing local data is typically much faster than accessing remote data. A successful parallel programming environment should, first, make it clear to the programmer when data being accessed is local or

remote, and, second, as much as possible allow the processor to continue doing useful work while remote data are being accessed. Charm++ addresses the first issue by only allowing the programmer to access remote data by means of a method invocation on a remote object, thus making it clear to the programmer when local data are being accessed and when remote data are being accessed. Furthermore, Charm++ addresses the second issue by employing asynchronous method invocation mechanisms in which a method invocation on a remote object does not cause the calling process to block. Instead, the caller continues execution after the method invocation with the expectation that the remote process will respond, via an asynchronous method invocation of its own, when the result of the computation is available.

- Support for irregularly-structured problems – Several interesting scientific applications involve irregular problem structures. Examples of such applications include adaptive mesh refinement, irregular finite-element problems, and N-body simulations. Charm++ provides an attractive environment for developing software for these types of problems because it provides the ability to dynamically create processes for solving portions of a problem and the ability to dynamically load balance processes over the lifetime of a computation.

- Performance – In addition to the design issues listed above, a successful parallel programming system must provide good performance. The challenge, however, is that the goal of providing good performance is often at odds with other design goals such as portability and modularity. An ongoing concern for development of the Charm++ system is delivering good performance to the application layer.

This chapter describes some of the details of the Charm++ language and how they relate to these design goals.

3.1 Chares

Programs written in Charm++ consist of one or more parallel objects called *chares.* A chare is similar to a C++ object except that public entry methods within a chare may be accessed remotely from other processes. The only way to access the state of a chare is through its public interface, therefore a programmer is acutely aware of which data accesses are local and which are remote while at the same time having a simple and consistent mechanism for accessing remote data.

Every Charm++ program contains a chare, called the *main chare*, in which execution begins. The main chare is responsible for creating additional chares to carry out the work of the parallel computation. Chare creation is considered a relatively low-cost operation and is the primary mechanism by which Charm++ supports irregularly-structured computations. Furthermore, because chares can be mapped dynamically to different processors, the Charm++ system can dynamically balance loads across all processors.

Collections of chares are also available to the programmer. A *chare group* is a collection of chares in which exactly one chare is placed on each processor. A typical use of chare groups is for system-related applications in which the programmer specifically needs to perform some action on each processor in the system. In addition to chare groups, collections of chares called *chare arrays* are also available to the programmer. With chare arrays, an arbitrary number of chares is placed on each processor in the system. This distribution of chares across processors is based on the Charm++ load balancer and the mapping may be dynamically modified to optimize the computation during execution. Chare arrays are the typical way that collections of chares are used in Charm++ programs.

3.2 Message-Driven Execution

The Charm++ system automatically generates messages that are sent from the local chare to a remote chare when a remote method invocation takes place. As messages arrive on the

remote chare, they are placed in a queue of messages waiting to be processed. The Charm++ scheduler removes a message from the chare's message queue and invokes the appropriate method on the chare to handle the message. The scheduler allows the method to run to completion before selecting another message from the queue. In this way, program execution within a chare is sequential while program execution across chares takes place in parallel. Overall, the concept of executing software handlers in response to the availability of messages is called *message-driven execution.*

After a local chare issues a remote method invocation on a remote chare, execution on the local chare continues immediately. Stated in other words, execution on the local chare does not block waiting for the method on the remote chare to complete. If the local chare needs some result computed by the remote method invocation, that result will be communicated back to the local chare by an additional remote method invocation from the remote chare to the local chare. The fundamental benefit of making remote method invocations non-blocking is that it allows the programmer to overlap computation with communication during program execution. A traditional difficulty in message passing systems is that the time spent communicating among processes is essentially wasted time. By using a message-driven approach, Charm++ masks communication latency and allows useful computation to continue.

3.3 Converse

The Charm++ language itself is implemented on top of a runtime library called Converse [15]. The Converse framework is intended to serve as a portable foundation for higher-level language and library writers and as such supports several different programming styles. Converse provides features such as asynchronous message passing mechanisms, thread objects, a generalized scheduler that coordinates both the delivery of messages and the efforts of threads, and load balancing primitives. These features are implemented as modules, allow-

ing the language or library designer to use only those features that are necessary and exclude everything else.

Message passing in Converse adheres to Active Messages [27] semantics. Each Converse process registers one or more handler functions designed to receive an incoming message and perform some corresponding processing. When a sender sends a message to a receiver, it first sets a handler on the message. The handler function is automatically invoked on the receiver when the message arrives. It should be clear that these semantics are quite similar to the remote method invocation used by the Charm++ system itself.

Figure 3.1 shows the structure of a parallel application that uses Charm++. The application itself is written in terms of Charm++ chares that interact with each other via asynchronous remote method invocations. These remote method invocations are translated by the Charm++ compiler into calls to Converse handlers. Finally, Converse passes the messages to the remote processor by means of a network transport within the Converse Machine Interface (CMI).

The CMI defines a minimal interface between the machine independent parts of Converse and the parts that are unique to various target platforms. The CMI is responsible for process creation and termination, sending and receiving messages, and low-level utility functions such as wallclock timers and mutexes for accessing memory. CMI versions exist for target architectures that pass messages via UDP/IP (the Net version), via Message Passing Interface (the MPI version), and via Quadrics Elan (the Elan version), among others.

3.4 Charm++ Program Files

This section examines a simple "Hello World" program written in Charm++. In this program, chares arrange themselves into a ring structure. The main chare creates a chare array and then invokes a method to display a message on the first element of the chare array. After the first chare displays its message, it invokes a method on the second chare in the array to

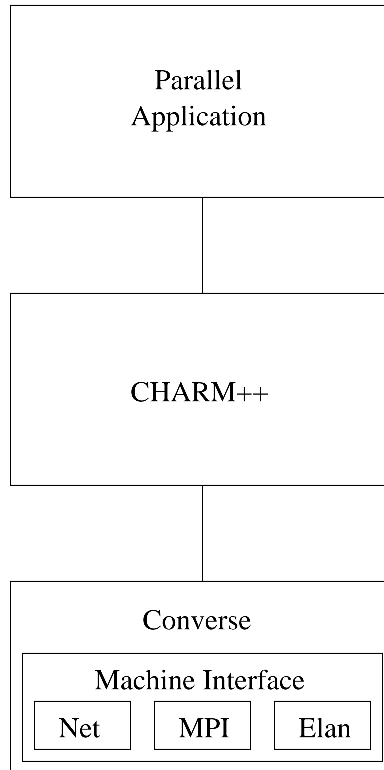


Figure 3.1: Structure of a parallel application that uses Charm++

display a message. This chain of method invocations continues until the last chare in the ring displays its message, at which time it invokes a method on the main chare to signal that the computation has completed.

Charm++ programs consist of three types of source code files [5]. Interface files are designated by the extension ".ci" and describe the main chare and any other chares used in the computation. Interface files also describe the public entry methods that may be invoked remotely on a chare. In addition to interface files, Charm++ programs consist of standard .h files and .C files used in traditional C++ programs.

Figure 3.2 shows the Hello.ci interface file for the Hello World example. The file defines three readonly variables (lines 3-6), two that represent *handles* that are essentially global pointers to the main chare and to the chare array, and one that maintains a count of the number of chares in the chare array. Readonly variables are written once and then the value written is available in all chares in the computation. Also defined within the Hello.ci

```
mainmodule Hello
{
  readonly CProxy_HelloMain mainProxy;
  readonly CProxy_HelloArray a;

  readonly int num;

  mainchare HelloMain
  {
    entry HelloMain ();
    entry void Finished (void);
  };

  array [1D] HelloArray
  {
    entry HelloArray (void);
    entry void PrintHello (void);
  };
};
```

Figure 3.2: The "Hello World" program .ci file

interface file is the main chare, HelloMain (lines 8-12), and the one-dimensional chare array, HelloArray (lines 14-18). Entry methods which are callable via remote method invocation are also defined within the main chare and the chare array.

Figure 3.3 shows the Hello.h header file for the Hello World example. This is a standard header file as used in C++. The main chare, HelloMain, inherits from the class Chare. Similarly, the chare array, HelloArray, inherits from the class ArrayElement1D.

Figure 3.4 shows the Hello.C source code file for the Hello World example. When the Charm++ compiler compiles the interface file, it generates the files "Hello.decl.h" and "Hello.def.h" to hold corresponding declarations and definitions for the chares used in the program. The programmer must include these files into the source code as shown in this example (lines 1 and 41).

The HelloMain constructor (lines 8-18) saves a handle to itself in a readonly variable (line 10) so that the chares in the chare array may later use it to communicate with the

```
class HelloMain : public Chare
{
  public:
    HelloMain (CkArgMsg *msg);
    void Finished (void);
};

class HelloArray : public ArrayElement1D
{
  public:
    HelloArray (void);
    HelloArray (CkMigrateMessage *msg);
    void PrintHello (void);
};
```

Figure 3.3: The "Hello World" program .h file

main chare. Next, it creates a chare array with a user-specified number of chares (line 16). Finally, it invokes the `PrintHello()` method on the first chare in the array to display a message (line 17). The first chare in the chare array displays a message (lines 31-33) and then invokes the `PrintHello()` method on the chare with the next higher array index (line 35). This process continues until the last chare in the chare array is reached, at which time the `Finished()` method is invoked on the main chare (line 37) and program execution ends.

3.5 Additional Features

In addition to the basic language features described in this chapter, Charm++ includes many additional features that are beyond the scope of this introduction. Examples of these features include chare arrays with two-dimensional, three-dimensional, and user-defined indexes; reduction operations, which perform a single operation such as add, max, or min over an entire chare array; prioritized execution, in which messages are tagged with a priority and then processed in priority-rank order; and a personalized collective communications library which provides optimized routines for performing all-to-all communications operations. Fur-

```
#include "Hello.decl.h"
#include "Hello.h"

CProxy_HelloMain mainProxy;
CProxy_HelloArray a;
int num;

HelloMain::HelloMain (CkArgMsg *msg)
{
  mainProxy = thishandle;
  if (msg->argc != 2) {
    ckout << "Usage: ./charmrun Hello <number of elements>" << endl;
    CkExit ();
  }
  num = atoi (msg->argv[1]);
  a = CProxy_HelloArray::ckNew (num);
  a[0].PrintHello ();
}

void HelloMain::Finished (void)
{
  ckout << "Main Chare is finished." << endl;
  CkExit ();
}

HelloArray::HelloArray (void) { }
HelloArray::HelloArray (CkMigrateMessage *msg) { }

void HelloArray::PrintHello (void)
{
  ckout << "Hello World from array element " << thisIndex
        << " on processor " << CkMyPe ()
        << " of " << CkNumPes () << " processors." << endl;
  if (thisIndex < (num-1)) {
    a[thisIndex+1].PrintHello ();
  } else {
    mainProxy.Finished ();
  }
}

#include "Hello.def.h"
```

Figure 3.4: The “Hello World” program .C file

thermore, companion tools provide features such as the ability to graphically analyze the performance of Charm++ programs to allow the programmer to better understand and optimize Charm++ programs.

Chapter 4

Virtual Machine Interface

As low-latency high-bandwidth networks such as Gigabit Ethernet [13], Myrinet [1], and InfiniBand [22] have received increased usage in high-performance cluster environments, the focus has switched from hardware performance to the performance of the underlying messaging software. Delivering point-to-point communication performance near what is achievable from the raw network hardware is now the primary goal to message layer designers. Furthermore, messaging layers are now expected to address several secondary goals including portability, monitoring and management, and support for applications running in distributed grid-computing environments.

The Virtual Machine Interface (VMI) [21, 19] project at the National Center for Supercomputing Applications was created to address these primary and secondary goals within the context of NCSA high-performance computation clusters. At the time of this writing, VMI is currently in beta testing for its second major release, VMI 2.0. This version of the software provides several compelling features:

- Multiple interconnects – VMI is designed to provide a single programming interface to the various system area networks commonly used in high-performance commodity clusters. Software implemented on VMI immediately gains access to all of the network interconnects supported by VMI while paying a small overhead of only a few microseconds per message. Furthermore, the underlying network interconnect may be switched simply by changing the contents of a file that describes the devices used for

the computation; no recompilation or relinking of the application software is necessary.

- Data striping and automatic fail-over – Because VMI operates as a software layer directly above the native network interconnect layer, it can stripe data across multiple network interconnects, even if these network interconnects are heterogeneous. By striping data across multiple network interconnects, VMI can deliver the aggregate bandwidth available from all interconnects to the application. Furthermore, if one interconnect fails, VMI can simply continue operating with any remaining interconnects.

- Portability – VMI is designed to be portable in two ways. First, VMI is designed to be portable to a wide variety of network interconnects. The challenge to this goal is the difficulty in designing a single Application Program Interface that can encompass all of the lower-level network APIs currently available while simultaneously providing excellent performance by giving the programmer access to some of the unique features of individual interfaces. Second, VMI is designed to be portable to a wide variety of platforms. Currently, VMI is available on both IA-32 and IA-64 architectures. Additional architectures such as Alpha and Power4 are currently being considered.

- Scalability – As high-performance commodity clusters increase in popularity, there is also a trend toward an increase in the number of nodes in a single cluster. Clusters with hundreds or thousands of nodes are now common. To this end, VMI is designed to scale to upward of several thousand nodes. The most critical way this is accomplished is by ensuring that none of the algorithms used within VMI have linear complexities with regards to the number of nodes in a computation.

- Support for distributed grid-based computing – As high-performance commodity clusters increase in popularity, there is a growing desire to connect multiple clusters together in order to harness the aggregate power of all machines. The challenges to this goal are twofold. First, the messaging layer must scale to hundreds or thousands of

nodes, just like in the case of building independent clusters that each contain a large number of nodes. Second, the messaging layer must not only provide good performance for the variety of system area networks used within each cluster but must also provide good performance for the local area networks and wide area networks used to connect the clusters themselves together. Design decisions regarding bandwidth and latency, for example, may be applicable to system area networks but not to the wide area. Because VMI is designed to be scalable to upward of several thousand nodes and because it readily supports various types of interconnects, it is a favorable platform for distributed grid-based computing. Furthermore, algorithms within VMI are designed to be latency tolerant in order to allow VMI to function correctly over wide area networks.

- Dynamic monitoring and management – Adding features to a messaging layer such as support for data striping and automatic fail-over, support for upward of thousands of nodes, and support for distributed grid-based computing adds complexity to the messaging layer. In order to deal with this added complexity, the messaging layer needs to support capabilities for dynamic monitoring and management. VMI includes capabilities for monitoring the state of the messaging layer in real time and dynamically managing the state of the stack.

This chapter provides an in-depth description of the Virtual Machine Interface design and a discussion of how the VMI design achieves the goals outlined above.

4.1 VMI Network Stack

Figure 4.1 shows the structure of a VMI application. The application is typically written in terms of an intermediate language or library rather than directly on top of VMI itself since VMI provides only simple point-to-point communication primitives which are an ideal target for language or library developers but generally too low level to be appropriate for application

developers. An example of such an intermediate language or library is the Message Passing Interface described in Chapter 2.

Below the VMI core is a set of dynamically-loaded device driver modules that can filter data or pass it to the network. These modules are organized into device *chains*. At a minimum, each VMI process must contain two chains: a *send chain*, which processes data sent to the network, and a *receive chain*, which processes data received from the network. An *alert chain* may also may be defined, and the VMI core will use this chain to signal error conditions. Finally, additional device chains may be defined for use by the intermediate language or library as desired.

The send chain shown in the example in Figure 4.1 defines devices that filter the data by encrypting and compressing it and then finally send the data over a TCP/IP network. The corresponding receive chain receives data from the TCP/IP network and filters it by decompressing and decrypting it before finally delivering the data to the process.

A specification file written in XML [3] describes the device drivers and chains used for a VMI program. For each chain, the devices on the chain and the order of these devices is described. The devices used for a program can be changed easily by modifying the specification file; no recompiling or relinking of the application is required. For example, simply by editing a few lines of XML, a VMI program could be changed from using a TCP/IP network transport to using Myrinet.

4.2 I/O Request Blocks

Data and control messages are passed between devices on a chain using *I/O Request Blocks* (IRBs). Each IRB contains a pointer to a stack, a status field, a pointer to the connection that the IRB is associated with, and other miscellaneous pieces of state. The stack, in turn, encapsulates the state of each device on the chain that the IRB is sent on, with one entry on the stack per device on the chain. Figure 4.2 provides a diagram showing the structure

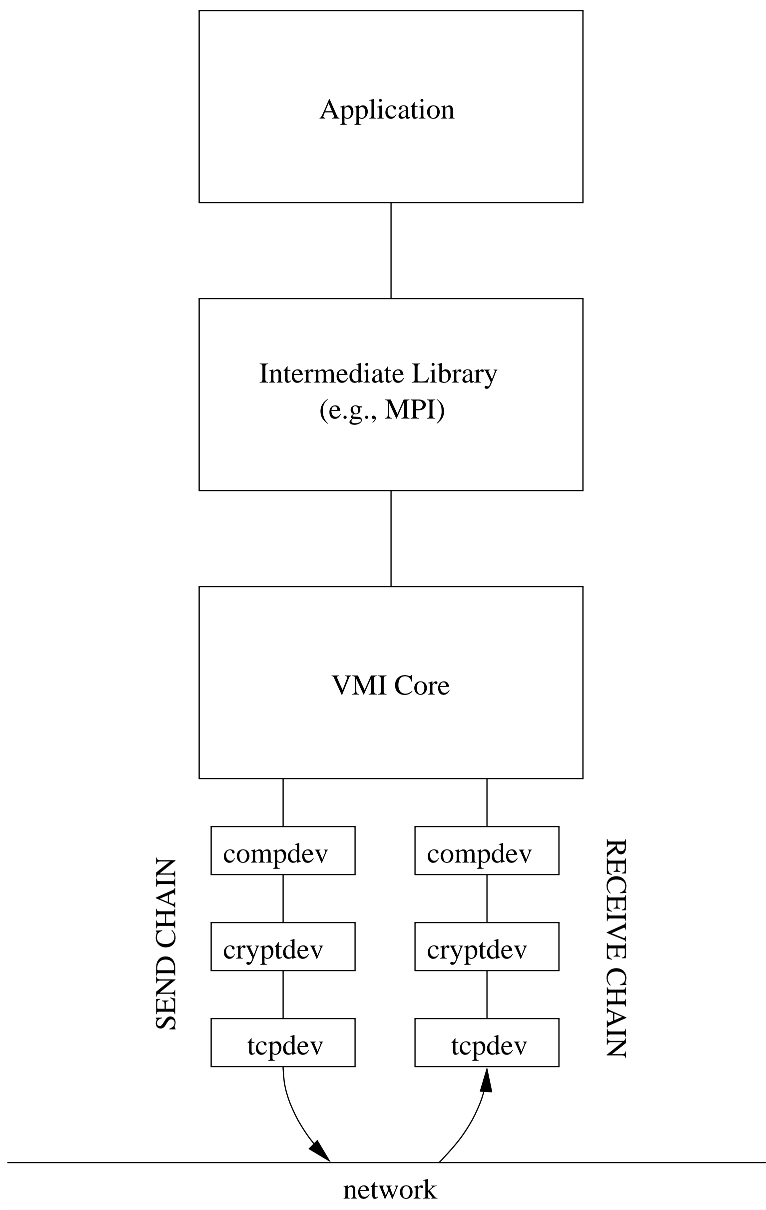


Figure 4.1: Structure of an application that uses VMI

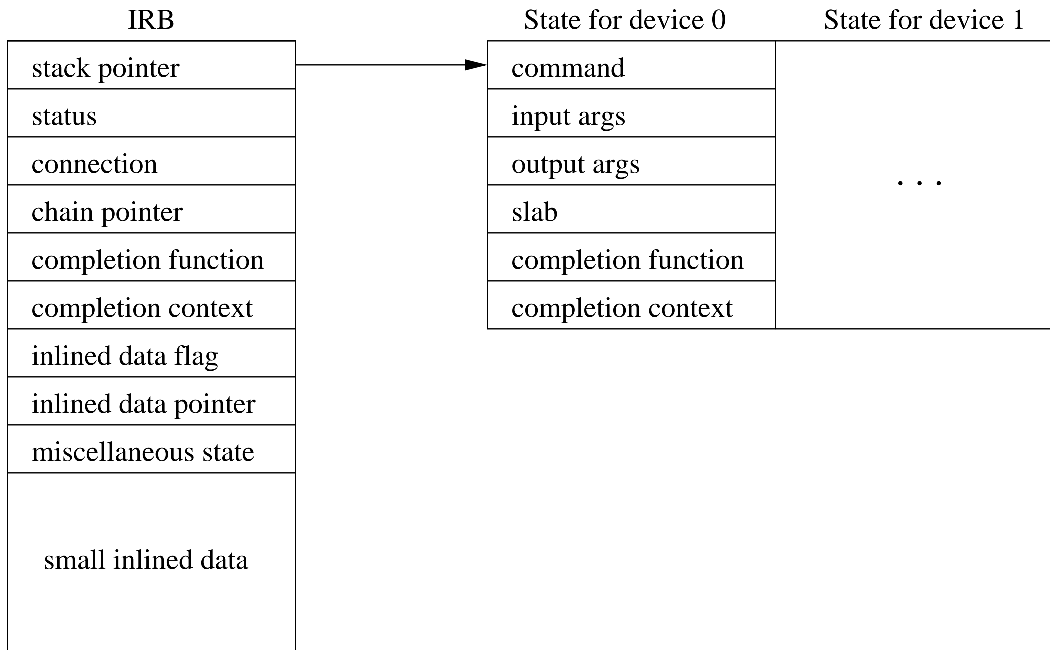


Figure 4.2: Structure of an I/O Request Block (IRB)

of an IRB.

Each element on the stack contains a structure consisting of several fields. The command field specifies the command corresponding to the IRB. Table 4.1 provides a list of possible commands and their meanings. The slab field is used to hold the message data that is passed to the network. The input args and output args fields are used to hold argument data to and from the device. Finally, the completion function and completion context fields are used when the last device on a chain completes processing of the IRB. When the last device on a chain finishes with an IRB, it invokes the completion function and passes the completion context, an arbitrary pointer to user data, to the function as an argument.

The last device on a send chain is known as a *sink device*. When an IRB reaches a sink device, the device has two options for handling the IRB. If the command contained in the IRB can be completed immediately, the device *completes* the IRB, causing each device on the IRB stack to be popped and the completion functions called in order. On the other hand, if the command contained in the IRB cannot be completed immediately, the device *pends* the IRB. A connection to a remote peer, for example, is likely to involve processing delay while

Command	Meaning
Attach	Attach a device to the specified chain
Detach	Detach a device from the specified chain
Connect	Open a connection to the specified peer
Disconnect	Close a connection to the specified peer
Connect Request	The specified peer is attempting to open a connection
Disconnect Request	The specified peer is attempting to close a connection
Send	Send data on the specified connection
Receive	Data has been received on the specified connection
Alert	An error condition is present on the device

Table 4.1: IRB commands and their meanings

the connection is opened. When a pended IRB finally completes, the IRB stack is popped and the device completion functions are invoked. In this way, all data in VMI is treated asynchronously and the intermediate language or library can readily allow the application to continue to make progress even when some communication channels are stalled.

4.3 Message Streams

Messages are transmitted between VMI peers as datagrams on a unidirectional path called a *stream*. Each stream is associated with a bidirectional *connection* between two peers. At most, one connection binds each pair of peers while any number of streams may be associated with each connection.

Message data are encapsulated into one or more *slabs* which travel along a stream. In the event that message data must be fragmented, for example in the case of an underlying network with a smaller Maximum Transmission Unit (MTU) size than the message data size, the extra data are put into an additional slab. Data within a single slab arrive in-order at the receiving process, but slabs may arrive out-of-order relative to one another.

When a slab arrives at a receiving process, the process has two options for handling the slab. If the receiving process has enough resources to process the slab data immediately, it handles the message data and then tells VMI that it is done with the slab. This enables

VMI to release memory resources consumed by the slab. If, on the other hand, the receiving process cannot handle the slab data immediately, it tells VMI that it wishes to *grab* the slab. VMI then transfers ownership of the slab to the process. At such time when the process has sufficient resources to handle the data contained within the slab, it does so and then tells VMI to *release* the slab, freeing its memory. The benefit of such functionality is that it allows the process to avoid copying the data to a temporary buffer when it cannot process the data immediately.

Most high-performance messaging layers require data to be registered before being used in a communication operation. The registration operation typically pins the data into physical memory, and this implies that the data must be in a contiguous block of memory. VMI uses *buffers* to represent contiguous blocks of memory that are suitable for registration with a lower-level messaging layer. Buffer registration is considered a relatively expensive operation. Furthermore, most messaging layers require that registered memory be page-aligned and of a size that is a multiple of the operating system page size. Thus, it is inefficient to register each piece of communication data independently. VMI provides a structure called a *buffer op* which represents a contiguous block of memory within a buffer. When a VMI program starts, VMI registers several buffers with the underlying messaging layer. As data are sent over the network, these buffers are divided into buffer ops to hold message data. When buffer ops are no longer needed, they are returned to the pool of pinned buffer memory.

The data in a slab are contained within one or more buffer ops that may each be dis-contiguous from one another. To optimize performance, VMI provides the ability to send and receive data from dis-contiguous regions with gather/scatter operations. These operations are implemented efficiently by simply adding or removing buffer ops from a slab.

Figure 4.3 shows the organization of streams, slabs, buffers, and buffer ops.

The VMI API provides two ways for the programmer to send messages on a stream. First, programmers may use the function `VMI_Stream_Send_Inline()` to perform an inline send of message data on a stream. In an inline send, the message data are copied into an

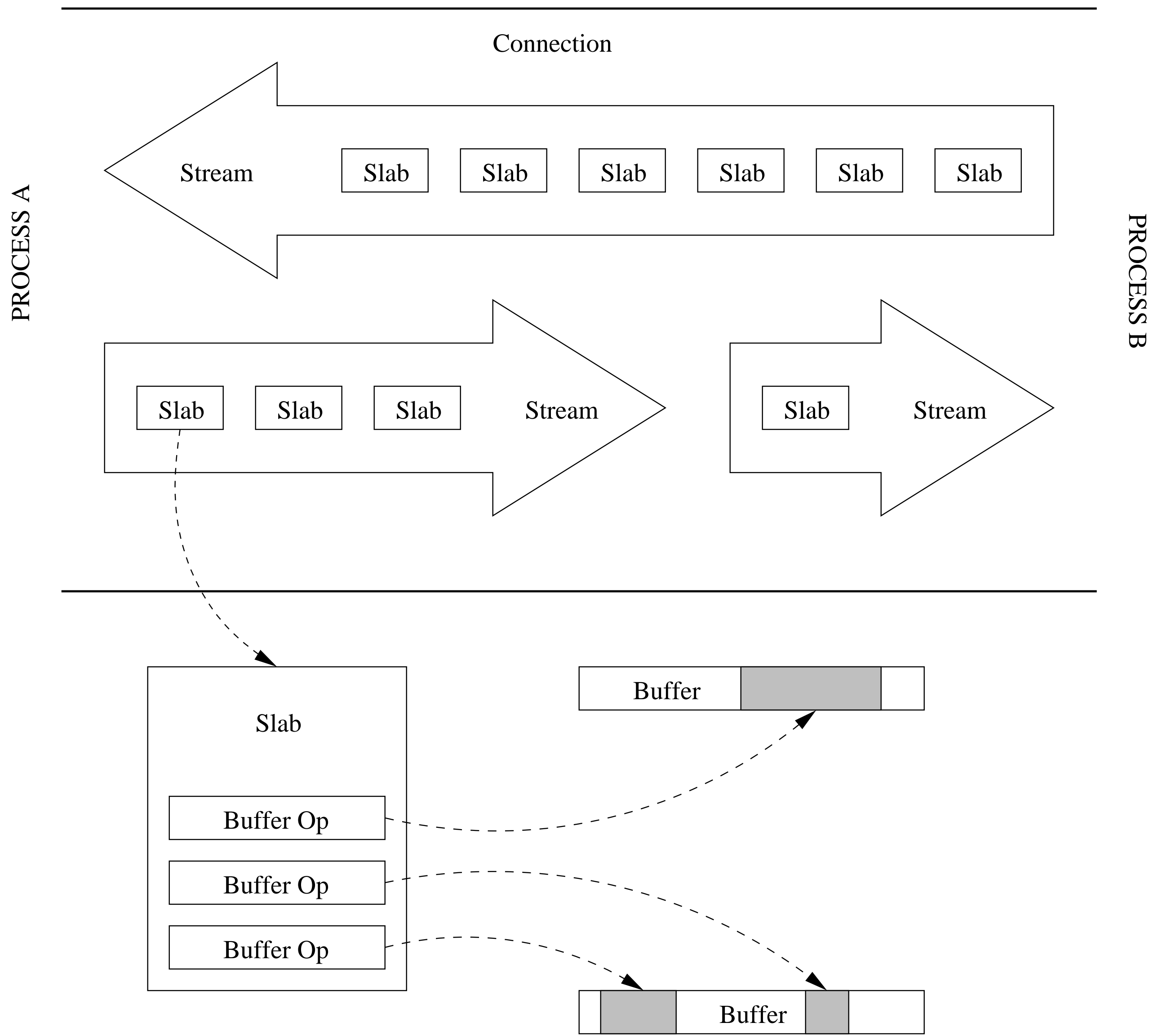


Figure 4.3: Structure of VMI Stream, Slab, Buffer Op, and Buffer [19]

inlined data area in the IRB associated with the message (recall Figure 4.2). The inlined send function is synchronous, causing the caller to block while message data are copied into the IRB and the IRB is dispatched onto the network. For small messages, it is more efficient to copy the message data and wait for them to be dispatched to the network. As the size of message data increases, it becomes less efficient to copy the message data. To this end, the VMI API provides a second mechanism for programmers to send messages on streams. Using this second method, the programmer first obtains a pinned region of memory for the message data either by calling `VMI_Buffer_Allocate()`, which obtains the pinned region by registering it dynamically, or by calling `VMI_Cache_Register()`, which obtains a pinned region from a large pool of pre-registered memory maintained by VMI. Next, the programmer calls `VMI_Stream_Send()` to send the message data held in the pinned buffer on a stream. In this call, the programmer may request that a *completion function* be invoked with a *completion context*, an arbitrary pointer to user memory, passed to the completion function. The stream send is dispatched asynchronously and control returns immediately to the caller. When the send completes, VMI automatically invokes the completion function to enable the programmer to deallocate the pinned memory used for the send.

4.4 Remote DMA

In addition to supporting message sends via streams, VMI supports message sends via Remote DMA (RDMA). When using RDMA to send messages, processes access the address spaces of other processes directly to write or read message data. No interaction by the remote process is necessary. Furthermore, by accessing the remote process's address space directly, no buffering is required for transferring the message data from the network adapter's memory to the process's memory. These zero-copy receives greatly improve the latency of message passing operations.

The VMI implementation of RDMA supports only put operations for writing data into

a remote process's memory. VMI drivers for network transports that have direct support for RDMA, such as Myrinet or InfiniBand, utilize the underlying network's RDMA mechanisms. For network transports that do not directly support RDMA, VMI emulates RDMA by using stream sends. In either case, the programmer uses a common API for accessing VMI RDMA functionality regardless of whether RDMA is directly supported or emulated on the underlying network hardware.

To use RDMA, the sending process first calls `VMI_RDMA_Set_Publish_Callback()` to register a *publish callback function* for a given connection. When the receiving process on that connection publishes a region of memory for the sender to put message data into, VMI automatically invokes the publish callback function and passes a pointer into the receiving process's address space. The sending process uses this pointer in a call to `VMI_RDMA_Put()` to write the message data directly into the receiving process's memory space. In the call to `VMI_RDMA_Put()`, the programmer can specify a *put completion function* along with a *put completion context*. When the RDMA put operation completes, VMI automatically invokes the put completion function and passes in the put completion context. The sending process can then deallocate any state that was used during the put operation.

The receiving process performs a similar set of operations to use RDMA. First, the receiving process calls `VMI_RDMA_Set_Notification_Callback()` to register a *notification callback function* for a given connection. When an RDMA put from the remote process completes, VMI automatically invokes the notification callback function to inform the receiver that new message data are available. Next, the receiving process must pin down a registered region of memory by calling either `VMI_Allocate_Buffer()` to dynamically register the memory region or `VMI_Cache_Register()` to obtain the memory region from a large pool of pre-registered memory maintained by VMI. The receiver then publishes the address of the pinned region of memory for the sender to put message data into by calling the function `VMI_RDMA_Publish_Buffer()`. As previously described, when a put from the sending process completes, VMI invokes the notification callback function, passing the address of the memory

Device Name	Description
hdralert	attaches a header consisting of machine name and date to alert messages
loopback	delivers message data to the underlying messaging hardware's loopback
mst-vapi	provides support for InfiniBand hardware
myrinet	provides support for Myrinet hardware
passthru	simply passes data through untouched
selfdev	handles message data where the source and destination are the same
shmem	sends message via shared memory to a process within same SMP node
syslog	logs an alert message to the Unix system log
tcpdev	sends messages via TCP/IP

Table 4.2: VMI supported devices

region holding the message data just received.

4.5 Loadable Devices

The VMI 2.0 release[1] includes several device modules classified as supported devices. These devices, along with a short description of each device, are listed in Table 4.2. Among these, the devices that support communication on TCP/IP, Myrinet, and InfiniBand are perhaps the most important.

In addition to the supported devices, the software distribution includes several experimental devices. Due to the modular nature of VMI, developing new devices can be readily done by individuals working independently from the main VMI development team. Examples of such efforts are the compression device and the encryption device described in [20]. Experimental devices that are determined to address a need with a broad scope will be incorporated into the supported devices of the VMI distribution after passing through extensive regression testing. Table 4.3 lists the current experimental devices included in the VMI software distribution.

[1] At the time of this writing, the current release is VMI 2.0 Beta2, released July 2, 2003.

Device Name	Description
aadev	plays audio as various types of IRBs travel down the send chain
compdev	compresses and uncompresses message data
cryptdev	encrypts and decrypts message data
fsdev	sends messages via files on a shared filesystem
profiledev	creates a profile of all message data sent by a process
reorder	reorders data received by a process so it appears to arrive in-order
skel	an example skeleton device useful to developers of new devices
via	provides support for VIA [?] transport
xfer	branches the send chain to allow attachment of multiple sink devices

Table 4.3: VMI experimental devices

4.6 Monitoring and Management Framework

The VMI remote monitoring and management facilities enable the state of the messaging layer to be monitored in real time. Based on the information gathered from monitoring, the messaging layer may also be managed dynamically. The granularity of this management capability ranges from the ability to modify individual parameters for lower-level network devices to the ability to terminate entire processes running within the computation. VMI includes a modular library that provides a simple Application Program Interface to this functionality.

In addition to the library that provides access to the monitoring and management framework to application software, the monitoring and management framework uses three daemon processes. Because VMI is designed with the intent of scaling to distributed grid-computing environments, these daemon processes are organized as a hierarchy. At the base of the hierarchy is the *VMIeyes* daemon. One VMIeyes daemon exists per node, and the purpose of this daemon is to track each VMI process running on the node and the devices used by each process. VMIeyes passes this information to the *Reaper* daemon at the next level of the hierarchy. One Reaper daemon exists per cluster. The Reaper keeps track of the state of all nodes in the cluster. Additionally, any management messages sent to a node originate from the Reaper and are communicated to the VMIeyes daemon running on an individual

node. Finally, at the highest level of the hierarchy is the *Nark* daemon. One Nark exists per grid, and the purpose of this daemon is to keep track of all Reaper daemons across an entire distributed grid-computing environment.

Chapter 5

Implementation Details

The efficient implementation of Charm++ on Virtual Machine Interface is deployed as a software module at the Converse Machine Interface (CMI) layer. The implementation is composed of some 4,000 lines of source code. Due to the highly modular design of Converse, only minor modifications to software outside the CMI layer are necessary. These modifications total approximately ten lines of code and are contained within the Converse core memory allocation routines. No modifications to the Charm++ source code are necessary. Figure 5.1 illustrates the relationship of the software required for the implementation with regard to the other software in a Charm++ application.

The remainder of this chapter presents details of the implementation and describes the optimizations used to provide good performance. Very broadly, these optimizations fall into the categories of optimizations to the message send path, optimizations to the message receive path, and the efficient management of memory resources.

5.1 Program Startup

Program startup in a parallel computing environment is often a very involved process. This is particularly true in the case of the efficient implementation of Charm++ on VMI. Starting Charm++ on VMI involves three primary operations. First, ranks are assigned to each process in the computation. Second, the VMI runtime system is initialized. Third, connections are

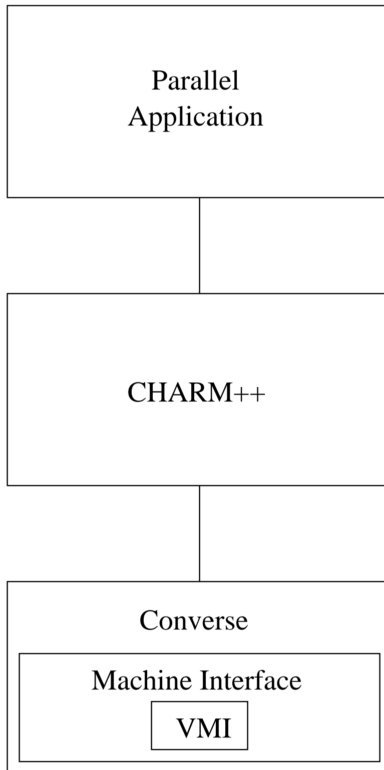


Figure 5.1: Structure of the efficient implementation of Charm++ on VMI

established among all of the processes in the computation. This section describes details of these three startup steps.

The implementation of Charm++ on VMI makes the assumption that each process allocated to a parallel job executes the Converse startup function `ConverseInit()`. The first thing that this function does is to establish the rank of each process in the computation. To do this, each process opens a network connection to a process called the *CRM* running on an arbitrary network-accessible machine. Each process belonging to a given job registers with the CRM, providing a job-specific *key* and an expected count of the number of processes for the job. The CRM groups each registration request that specifies the same key into a single job. After the specified number of processes for the job register with the CRM, it returns an ordered list of the processes in the job to each process. By examining this list, each process can determine its rank in the job along with the ranks of all other processes.

After establishing ranks, the `ConverseInit()` function must next initialize the VMI run-

time system within each process. To do this, each process examines its environment variable space to determine whether any VMI default behaviors are being overridden through values specified in runtime environment variables. Taking both default values and overridden values into account, the function `VMI_Init()` is invoked to initialize VMI. Immediately after VMI is initialized, various VMI-related dependencies are performed, such as the registration of buffer pools and a stream receive function.

Finally, in the third phase of program startup, connections are opened among all processes in the computation. Because VMI is a connection-oriented communication layer, each pair of processes must have a connection in order to communicate with each other. However, a connection is bi-directional, so each pair of processes requires only a single connection to communicate with each other. To open all required connections efficiently and without race conditions, each Charm++ on VMI process issues connection requests to all processes with a rank lower that that process's rank and waits for incoming connection requests from processes with a higher rank.

After the program is successfully initialized in each process in the computation, the Converse scheduler is started in each process and execution of the user's program code begins.

5.2 Message Sends

The Converse Machine Interface provides three low-level functions for sending data. Each function accepts the destination process that is to receive the message, the size of the message, and a pointer to the message data. The `CmiSyncSend()` function sends the message synchronously and returns to the caller only after the message data have been dispatched to the network. The `CmiAsyncSend()` function sends the message asynchronously and returns to the caller immediately. Additionally, this function returns a *CmiCommHandle* to the caller which the caller can use to determine when the asynchronous send completes. The contract between the caller and the CMI restricts the caller from modifying the message data

until an asynchronous message has been successfully dispatched to the network. Finally, the `CmiFreeSend()` function transfers ownership of the message data to the CMI. Sometime after the send completes, the CMI is expected to deallocate the memory consumed by the message buffer.

The CMI also provides message broadcast functions that provide functionality congruent to the send functions. The `CmiSyncBroadcast()` function broadcasts the message synchronously and returns to the caller only after the message data have been dispatched to all processes. The `CmiAsyncBroadcast()` function broadcasts the message asynchronously and returns immediately to the caller, passing back a CmiCommHandle that the caller can use to determine when all sends within the process complete. Finally, the `CmiFreeBroadcast()` function is used to pass ownership of the message data to the CMI and the memory consumed by the message buffer is deallocated after the broadcast completes. For each broadcast function, data are delivered to all processes in the computation *except the sending process.* If the caller wishes also to deliver the broadcast message to itself, variations of the broadcast functions exist.

The wide variety of send and broadcast functions provides a rich space for optimization to the machine layer developer. For example, any of the asynchronous send or broadcast functions can be implemented in terms of synchronous operations. Typically, synchronous operations provide less performance to the calling software layer because they cause the caller to block until the operation completes. The trade-off, however, is that synchronous operations are typically much easier to implement. Fortunately, VMI offers a framework that makes implementing asynchronous operations easy without sacrificing performance.

A challenge to writing asynchronous messaging layers is that each asynchronous send or broadcast operation typically requires some amount of state to be created and maintained during the lifetime of the operation. At a minimum, this state generally must contain a pointer to the message buffer used for the asynchronous operation so the memory for the buffer can be deallocated at the end of the operation. To further complicate matters, when

a system has multiple outstanding asynchronous operations, the associated state for a given operation must be located among the states for all outstanding operations.

Recall that VMI allows the state of an asynchronous operation to be maintained and accessed in a straightforward manner. Each asynchronous send operation allows the programmer to specify a completion function and a completion context. The completion context is simply a pointer to an arbitrary user-defined address in memory. When the asynchronous operation completes, VMI invokes the completion function and passes the completion context as an argument. The completion function can then perform any necessary deallocation and cleanup operations.

The implementation of Charm++ on VMI defines a structure called a *send handle* which is used internally to hold the state of all asynchronous send operations. A send handle contains a reference count, a pointer to the message buffer, and the size of the message data. The send handle is used as the completion context for VMI message sends; when an asynchronous message send completes, VMI passes the send handle corresponding to the completed send to the completion handler function.

The reference count field allows a single send handle structure to be used to implement both send and broadcast operations, and both synchronous and asynchronous variations of these operations. At the end of an asynchronous send operation, the completion function decrements the reference count field. If the reference count reaches zero, the completion function deallocates the message buffer within the handle and then deallocates the handle itself. Since asynchronous broadcasts are nothing more that multiple sends to different processes, the reference count for a broadcast operation is simply set to be equal to the number of processes to which the message is sent. When the last send in the broadcast completes, the reference count is decremented to zero and the completion function deallocates the message buffer and handle. Finally, synchronous operations can be trivially implemented by simply setting the reference count field to a value one higher than the number of send operations. The caller then waits until the reference count value reaches one, at which time

it knows that all other sends on the handle have completed. It is the responsibility of the caller to release the message buffer and send handle in the case of synchronous sends and broadcasts.

As described previously, a caller to asynchronous CMI functions such as `CmiAsyncSend()` receives a CmiCommHandle that can be used to determine when the asynchronous operation has completed. While it would be possible to simply return the send handle itself as the CmiCommHandle, this opens the possibility that the caller could somehow corrupt the internal state of the handle. To prevent this, the send handle contains an additional field, a pointer to a CmiCommHandle, which is used by the completion function to also update the state of the CmiCommHandle corresponding to the send.

Send operations are divided into three categories based on the size of the message data and implemented in a strategy best suited to the message. The *small message strategy* and *medium message strategy* are optimized to favor latency and are implemented with VMI streams. The *large message strategy* is optimized to favor bandwidth and is implemented with VMI RDMA. The message size boundaries between strategies can be changed at runtime. The default boundary size between the short message strategy and the medium message strategy is 512 bytes, and the default boundary size between the medium message strategy and the large message strategy is 4,096 bytes.

Both the small message strategy and the medium message strategy employ VMI streams. Small messages are sent with a VMI inlined send operation which copies the message data into the IRB and performs a synchronous stream send. Since the send completes synchronously, no completion function is invoked. Memory bandwidth is high enough on most machines that copying the data for small messages and dispatching the copy to the network is favorable. In contrast, the medium message strategy first pins the message data into memory and then allocates a send handle and performs an asynchronous stream send. When the asynchronous send completes, a completion function is invoked and the send handle is passed to the function, allowing the message data to be unpinned and state of the send to be deallocated. For

medium-sized messages, paying the cost of pinning memory used by the send operation and invoking a completion function to unpin the memory after the send completes is favorable.

The large message strategy employs a rendezvous protocol to set up VMI RDMA. An RDMA operation requires the receiver to publish the address of a pinned region of memory in which the message data will be deposited by the sender. It is infeasible for each process to maintain a separate pinned memory region for every other process in a computation. Instead, when a sender wants to use the large message strategy, it first sends a small message on a VMI stream to the receiver to signal the upcoming RDMA operation. Upon receipt of this message, the receiver pins a region of memory and publishes the address of this region to the sender. The sender then puts the message data directly into the receiver's address space. To avoid requiring the receiver to pin unusually large amounts of memory, which becomes infeasible as the size of a message grows, a maximum message chunk size for an RDMA operation is defined. To send a message that is larger than the maximum chunk size, multiple puts to the receiver are required. After each put, the receiver moves the chunk data into another region of memory and acknowledges to the sender that it has done so, allowing the sender to put the next chunk of data into the receiver's address space. For large-sized messages, sacrificing latency for the increased bandwidth available from RDMA is favorable.

Finally, two optimizations that are independent of VMI are used to improve the performance of the message send path. The first optimization involves the case where a process sends a message to itself. Self-sends happen in all of the BroadcastAll functions. Furthermore, a programmer may use a self-send to specifically invoke the handler function associated with a particular message on the local processor. To optimize message sends where the sending process and destination process are the same, it is important not to pass the message to the network hardware at all. To this end, each CMI send and broadcast function specifically checks for a self-send and enqueues such messages in a queue of local messages. The Converse scheduler checks the local message queue in addition to the remote message queue when searching for messages to deliver to the local process.

The second optimization that is independent of VMI is the use of spanning trees for optimizing broadcasts. Broadcasts that do not use spanning trees must send an individual message to each other process in a computation. As the number of processes in a computation increase, sending messages to each process becomes prohibitive. In contrast, broadcasts implemented in terms of spanning trees amortize sends across the other processes in the computation. Each process in the tree sends broadcast messages only to its *spanning neighbors*. These neighbor processes each forward the message to their spanning neighbors. Forwarding continues until the message reaches every process in the computation. The number of messages each process is required to send is bounded by the *spanning factor* of the tree. The default spanning factor used for the implementation of Charm++ on VMI is four, and this spanning factor can be changed by recompiling the CMI module.

5.3 Message Receives

In addition to providing a convenient framework for implementing efficient asynchronous message sends, VMI also provides features that enable the developer to easily write efficient message receives. Both stream receives and RDMA receives take place asynchronously. When a VMI program starts, it registers a handler function which is invoked asynchronously any time data are received on a stream for any connection. That is, a single handler function processes all stream data coming into the program. For RDMA data, a VMI program registers independent handler functions on a per-connection basis. For the Charm++ implementation on VMI, a single handler function is used to process all RDMA data coming into the program for all connections.

Since messages sent between two processes are dispatched via stream sends or RDMA independently, they may arrive at the receiver in arbitrary order. Because the Converse Machine Interface does not make any guarantees regarding message ordering but instead leaves ordering up to the application, the machine layer is free to place received messages

into the remote message queue immediately after they are completed. This greatly simplifies the design of the CMI for Charm++ on VMI because message reordering does not have to be implemented.

5.4 Memory Management

The final category of optimizations employed by the implementation of Charm++ on VMI involves efficient memory management. Repeated allocation and deallocation of memory is expensive in terms of machine cycles. Such operations occur within the Converse Machine Interface layer, for example, each time a message is received from the network. Furthermore, memory may be allocated and deallocated at the application level as the application creates and deletes Charm++ objects. To help avoid these costly operations, VMI provides facilities for managing memory pools. The implementation of Charm++ on VMI uses these facilities by maintaining five buffer pools for memory regions of sizes of less than 1,024; 2,048; 4,096; 8,192; and 16,384 bytes respectively. When a memory allocation operation is performed, the allocation is ultimately processed by the `CmiAlloc()` function. Calls to this function are intercepted and memory allocation requests are fulfilled by returning a block of memory from the next-higher-sized buffer pool. For example, a memory allocation request for 2,000 bytes would be fulfilled by allocating an entry from the second buffer pool, returning a block of memory of 2,048 bytes. The remaining 48 bytes in the buffer are wasted. Memory for allocation requests that are too large to fit into any of the buffer pools is obtained by a traditional call to `malloc()`. When memory is to be deallocated, the deallocation is processed by the `CmiFree()` function. Again, calls to this function are intercepted so that memory may be returned to the appropriate buffer pool. Internally, VMI allocates large blocks of memory for each buffer pool and manages the allocation and deallocation of memory from these pools.

A second area of memory management optimization that the implementation of Charm++

on VMI employs is efficient management of registered memory regions for send and receive operations. Typically, high-performance messaging layers such as Myrinet and InfiniBand require the memory used for send and receive operations to be pinned into core memory prior to the operation. Pinning the memory ensures that the contents of the memory region are not paged to disk by the operating system's virtual memory manager, allowing the network adapter to use Direct Memory Access to deposit or retrieve message data without interrupting the CPU. Registering and deregistering memory is a very expensive operation. Furthermore, the granularity of memory pinned by a registration operation is generally limited to page-sized regions. Upon initialization, VMI registers several pages of memory for send and receive operations and makes this pre-pinned memory available to higher level languages and libraries via API calls. The implementation of Charm++ on VMI uses this facility to improve performance.

Chapter 6

Performance

Evaluating the performance of the Charm++ implementation on VMI is critical to determine whether the project achieves its goal of giving Charm++ programs access to the features of VMI while incurring minimal overhead. The danger is that adding additional layers of software could potentially cause significant impacts to the performance of Charm++ applications. This chapter presents the results of the implementation running latency and bandwidth microbenchmarks and compares them to the performance of Charm++ implementations running on other communication layers.

6.1 Test Environment

Data for this thesis were collected on the Titan cluster [6] at the National Center for Supercomputing Applications from September 22, 2003 to September 26, 2003. Titan is a production-quality IA-64 architecture Linux cluster with a peak performance of one teraflop.

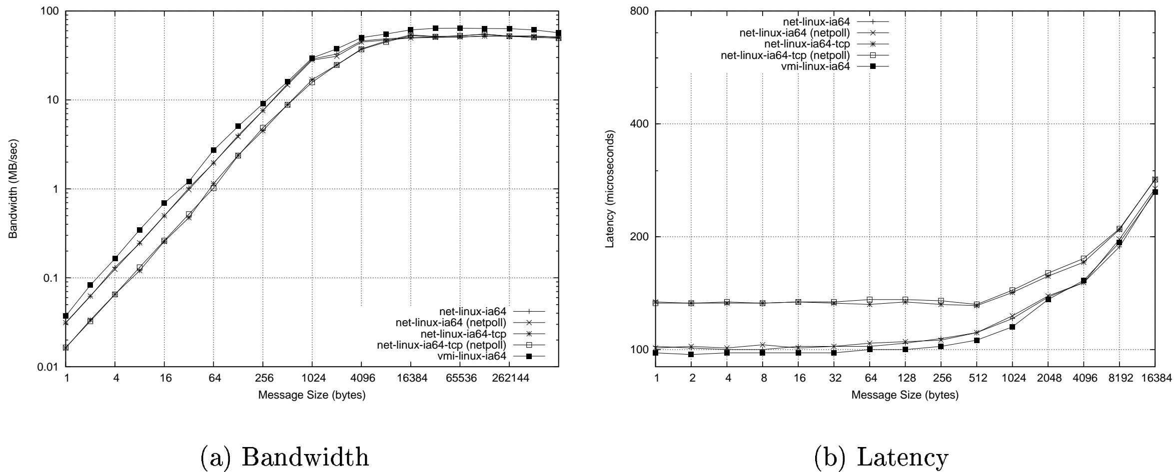
The Titan cluster consists of two access nodes, 128 compute nodes, and four storage nodes. Each access and compute node is a dual-processor IBM IntelliStation Z Pro 6894 workstation containing 800 MHz Itanium 1 (Merced) processors, four megabytes of L3 cache, and two gigabytes of ECC SDRAM. Access and storage nodes are connected to each other and to the compute nodes via Gigabit Ethernet. Compute nodes are connected to each other

via Gigabit Ethernet and Myrinet 2000 interconnects. All nodes run RedHat 7.1 with Linux kernel version 2.4.16. The compilers used on the cluster are the Gnu C/C++ compiler gcc version 2.96, and the Intel C/C++ compiler ecc version 7.0.

Performance data were collected for Converse and Charm++ running latency and bandwidth microbenchmarks for message payload sizes ranging from one byte to one megabyte in increasing powers of two. Data were collected for the net-linux-ia64, net-linux-tcp-ia64, and vmi-linux-ia64 versions of the Converse Machine Interface for Gigabit Ethernet and for the net-linux-gm-ia64 and vmi-linux-ia64 versions of CMI for Myrinet. Furthermore, the net-linux-ia64, net-linux-tcp-ia64, and net-linux-gm-ia64 versions can run in either *netpoll mode*, in which data are retrieved from the network via a polling operation, or in *SIGIO* mode in which data are retrieved from the network after an interrupt signal is raised. Performance data for both modes of these Charm++ versions were collected.

The compilations of Converse, Charm++, and the microbenchmark application code were done with the compile-time options `-O -DCMK_OPTIMIZE` for the net-linux-ia64, net-linux-tcp-ia64, and net-linux-gm-ia64 benchmark runs. For the case of the vmi-linux-ia64 benchmark runs, Converse and the microbenchmark application code were compiled with `-O -DCMK_OPTIMIZE` but Charm++ was compiled with only `-DCMK_OPTIMIZE`. Unfortunately when Charm++ on vmi-linux-ia64 is compiled with the `-DCMK_OPTIMIZE` compile-time option, incorrect code is produced and the resulting software produces segmentation faults when executed. This compile-time flag controls whether certain optimizations are included within the Charm++ source code.

The latency and bandwidth microbenchmarks both run eleven instances of the benchmark, discarding the first result to eliminate startup anomalies, and averaging the results of the remaining ten instances to produce a given benchmark number. For the latency microbenchmark, each test involves the sender sending 1,000 instances of message data of the requested size to the receiver and then waiting for an acknowledgment message after each instance. Time for each of the 1,000 instances is recorded and these times are averaged at



(a) Bandwidth

(b) Latency

Figure 6.1: Performance of Converse on Gigabit Ethernet

the end of the run to compute the latency in microseconds per message. For the bandwidth microbenchmark, each test involves the sender starting a timer, sending 1,000 instances of message data of the requested size to the receiver, waiting for a single acknowledgment from the receiver after all 1,000 messages are received, stopping the timer, and computing the bandwidth in megabytes per second.

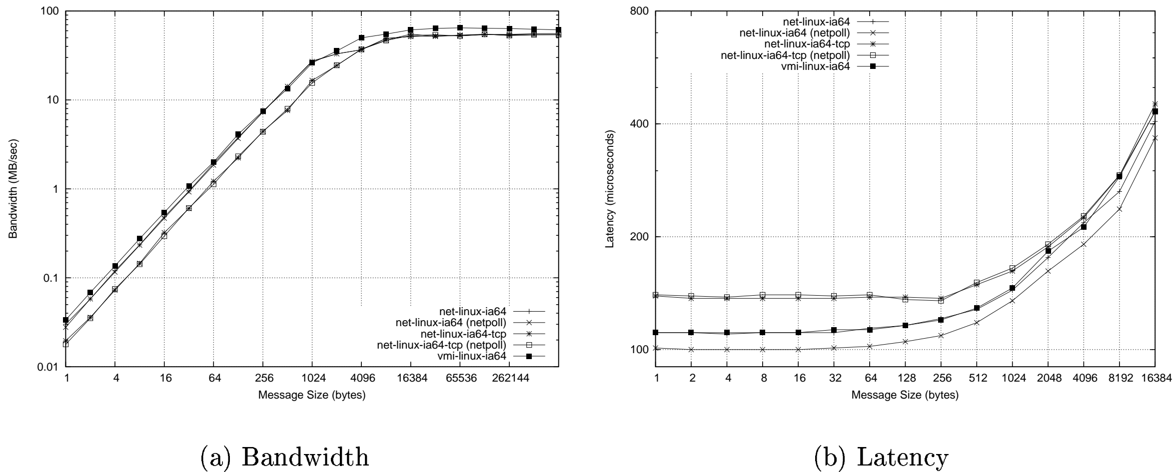
6.2 Gigabit Ethernet Performance

Figure 6.1 shows the performance of Converse on Gigabit Ethernet. Bandwidth is shown in Figure 6.1(a). The vmi-linux-ia64 version of Converse offers the most favorable performance, followed closely by the net-linux-ia64 version in both netpoll and SIGIO modes. Finally, the net-linux-ia64-tcp version, in both netpoll and SIGIO modes, offers the worst performance. The best bandwidth achieved by vmi-linux-ia64 is 64 megabytes/second. Similarly, latency is shown in Figure 6.1(b). Again, the performance of the vmi-linux-ia64 implementation is best, followed closely by both versions of net-linux-ia64. Both versions of net-linux-ia64-tcp are similarly matched and offer significantly worse performance. The best latency achieved by vmi-linux-ia64 is 97 microseconds/message one-way.

Figure 6.2 shows the performance of Charm++ on Gigabit Ethernet. Bandwidth is shown in Figure 6.2(a). Similar to the performance of Converse on Gigabit Ethernet, the vmi-linux-ia64 version of Charm++ offers the most favorable performance, followed closely by the net-linux-ia64 version in both netpoll and SIGIO modes. Finally, the net-linux-ia64-tcp version, in both netpoll and SIGIO modes is slowest. The best bandwidth achieved by vmi-linux-ia64 is 64 megabytes/second. While the performance difference between the net-linux-ia64 version and net-linux-ia64-tcp version remain similar between Converse and Charm++, the margin of improvement between vmi-linux-ia64 and net-linux-ia64 is more pronounced for Converse than for Charm++. This is most likely due to the inability to use the `-DCMK_OPTIMIZE` compile-time flag when building the vmi-linux-ia64 version of Charm++. Latency is shown in Figure 6.2(b). In this experiment, the net-linux-ia64 version of Charm++ running in netpoll mode offers the most favorable performance. The vmi-linux-ia64 and net-linux-ia64 (SIGIO) versions of Charm++ follow in performance and are very closely matched in relation to each other. Finally, both versions of the net-linux-ia64-tcp version of Charm++ offer the worst performance. The best latency achieved by vmi-linux-ia64 is 111 microseconds/message one-way. The performance of vmi-linux-ia64 and net-linux-ia64 (netpoll) versions relative to each other suggest that the above discussion regarding compile-time options for vmi-linux-ia64 may indeed be the source of its less impressive performance.

6.3 Myrinet Performance

Figure 6.3 shows the performance of Converse on Myrinet. Bandwidth is shown in Figure 6.3(a). The vmi-linux-ia64 version of Converse offers the most favorable performance followed by both versions of the net-linux-ia64-gm version. At 1,024 bytes, the net-linux-ia64-gm versions both show a dip in performance due to internal packetization of message data that is longer than 1,024 bytes. The best bandwidth achieved by vmi-linux-ia64 is 240 megabytes/second. Similarly, latency is show in Figure 6.3(b). Again, the performance of the



(a) Bandwidth (b) Latency

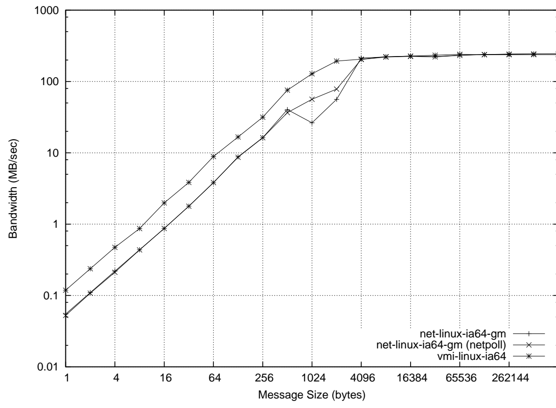
Figure 6.2: Performance of Charm++ on Gigabit Ethernet

vmi-linux-ia64 implementation is best, followed by both versions of net-linux-ia64-gm which are closely matched. The best latency achieved by vmi-linux-ia64 is 16 microseconds/message one-way.

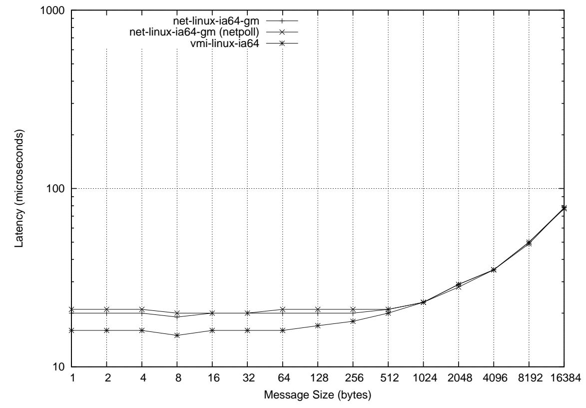
Figure 6.4 shows the performance of Charm++ on Myrinet. Bandwidth is shown in Figure 6.4(a). The vmi-linux-ia64 version of Charm++ offers the most favorable performance. Both versions of net-linux-ia64-gm follow significantly behind. Again, at 1,024 bytes, the internal packetization of net-linux-ia64-gm is apparent in performance. The best bandwidth achieved by vmi-linux-ia64 is 243 megabytes/second. Latency is shown in Figure 6.4(b). All three implementations of Charm++ show nearly identical performance on this benchmark. The best latency achieved by vmi-linux-ia64 is 20 microseconds/message one-way.

6.4 Interpretation of Results

Based on the performance graphs presented in the previous section, it seems safe to conclude that the implementation of Charm++ on VMI does indeed achieve the goal of making the features of VMI available to Charm++ programs without loss of performance. Of the eight graphs examined, performance of the vmi-linux-ia64 version is better than any of the

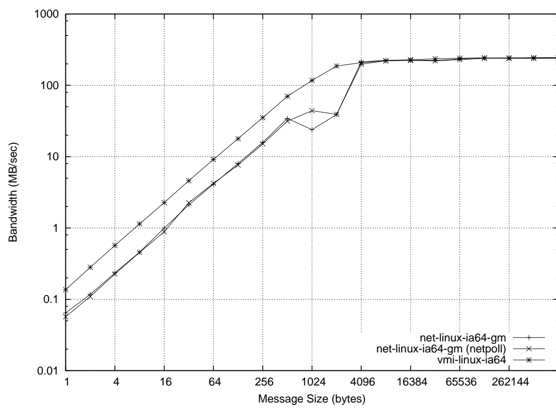


(a) Bandwidth

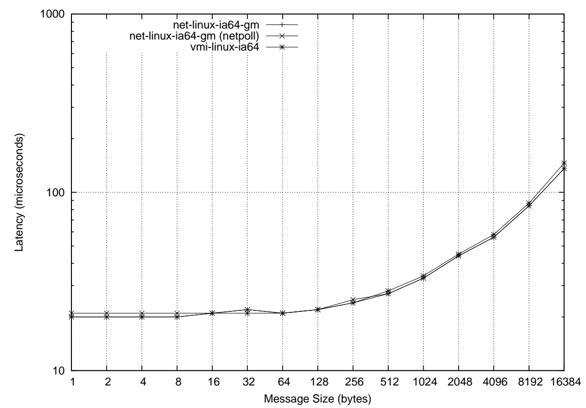


(b) Latency

Figure 6.3: Performance of Converse on Myrinet



(a) Bandwidth



(b) Latency

Figure 6.4: Performance of Charm++ on Myrinet

other implementations of Charm++ examined in this thesis in six cases, equal to the other implementations in one case, and worse than the other implementations in one case. An examination of the source code for the vmi-linux-ia64 Converse Machine Interface to determine the reason for segmentation faults in Charm++ when the implementation is built with the -DCMK_OPTIMIZE option is an important future point of investigation. After this problem is solved, it is expected that the performance of the efficient implementation of Charm++ on VMI will be better than all other implementations considered here.

Chapter 7

Conclusion and Future Work

This thesis has described an efficient implementation of Charm++ on Virtual Machine Interface and discussed the various design trade-offs involved. Performance of the implementation was evaluated for latency and bandwidth and compared to the performance of Charm++ implementations running on other communication layers. Based on this performance evaluation, it seems reasonable to conclude that the features provided by VMI have been made available to Charm++ developers with a minimum of overhead. In several cases, the performance of Charm++ on VMI is better than the performance of Charm++ on other communication layers that involve fewer levels of abstraction.

The efficient implementation of Charm++ on VMI is intended to be production-quality code with the potential for being deployed in environments such as the high-performance clusters at the National Center for Supercomputing Applications. To this end, future work on this code will focus on several key areas:

- Evaluation of performance on InfiniBand – The InfiniBand interconnect is expected to increase in popularity over the next several years and promises compelling performance in terms of latency and bandwidth. Performance of the efficient implementation of Charm++ on VMI should be evaluated on InfiniBand to ensure that the implementation is optimized for highly-efficient interconnects such as this.

- Application-specific benchmarking – Applications such as NAMD [18] include datasets

used to benchmark the performance of the software on various architectures. While the latency and bandwidth microbenchmarks used in this thesis are a good first step in evaluating the performance of the implementation of Charm++ on VMI, it is also important to evaluate the performance of the implementation with application-specific benchmarks due to the fact that they more clearly reflect the overall performance realized when handling real workloads.

- Comparison to performance of Charm++ mpi-linux version – A Converse Machine Interface layer exists for the successful Message Passing Interface (MPI) and can be used to build an mpi-linux version of Charm++. Additionally, a version of MPI running on VMI has been developed. Thus it is possible to deploy Charm++ on VMI via the use of the mpi-linux version. The performance of this version of Charm++ should be evaluated and compared to the performance of Charm++ implemented directly on VMI. Due to the nature of the mpi-linux version of Charm++, certain operations are inherently slow. For example, the only option the mpi-linux implementation has for determining whether message data are available from a remote process is to probe each MPI communication handle belonging to a given process. As the size of a computation increases, this probing scales poorly and limits the efficiency of the overall computation. For reasons such as this, the performance of the mpi-linux version of Charm++ running on VMI is expected to be much lower than the performance of the vmi-linux version.

- Implementation of shared memory support – An increasing number of commodity clusters employ Shared Memory Multiprocessor (SMP) systems with two or four processors per cluster node. The implementation of Charm++ on VMI supports SMP systems in two ways. First, messages sent between processes within the same SMP may be delivered via the network exactly like messages sent between processes on different machines. The disadvantage of this approach is that it involves unnecessary interaction with the network hardware, introducing needless latencies in message delivery. Second, mes-

sages sent between processes within the same SMP node may be handled by the VMI shared memory device. This approach has the advantage of eliminating the use of the network hardware, but still involves the overhead of four IRB traversals (send initiation, send completion, receive initiation, receive completion). The best approach is to build support for SMP directly into the vmi-linux Converse Machine Interface, similar to the support that exists in layers such as the net-linux CMI.

- Performance analysis of advanced VMI features – VMI provides several advanced features such as the ability to stripe data across multiple network interfaces and the ability to dynamically compress data passing over the network. Through the Charm++ implementation for VMI, these features are automatically available to Charm++ programs. Some effort should be spent evaluating the performance of these features.

- Evaluation of multi-cluster performance – As commodity cluster systems become ubiquitous, there is increasing interest in applications that can span multiple clusters to gain access to a larger numbers of processors than are available within the scope of a single cluster. The Charm++ implementation on VMI offers an attractive platform for deploying such applications because it abstracts details of the underlying network technologies used in each cluster. For example, an application that spans two clusters may need to use Myrinet for communication within one cluster, InfiniBand for communication within the second cluster, and Gigabit Ethernet for cross-cluster communication. The advantage offered by the implementation of Charm++ on VMI is that the VMI runtime handles details of the different networks and will use the most efficient communication path for sending messages between any arbitrary pair of processes. Evaluation of Charm++ on VMI in such multi-cluster environments is an important and interesting future endeavor.

References

[1] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King K. Su. Myrinet — A gigabit-per-second local-area-network. *IEEE Micro*, 15(1):29–36, February 1995.

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

[3] World Wide Web Consortium. Extensible markup language (XML) 1.0 (second edition) – W3C recommendation. http://www.w3.org/TR/2000/WD-xml-2e-20000814, 2000.

[4] Compaq Computer Corporation, Intel Corporation, and Microsoft Corporation. Virtual interface architecture specification. http://www.viarch.org/html/collateral/san_10.pdf, December 1997.

[5] Department of Computer Science,University of Illinois at Urbana-Champaign, Urbana, IL. *The CHARM++ Programming Language Manual*, 2003.

[6] National Center for Supercomputing Applications. Titan cluster homepage. http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/IA64LinuxCluster/.

[7] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, Department of Computer Science, University of Tennessee, April 1994.

[8] Ian Foster and Carl Kesselman. The globus toolkit. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 259–278. Morgan-Kaufmann, San Francisco, CA, 1999.

[9] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan-Kaufmann, July 1999.

[10] Ian Foster, Carl Kesselman, and Steven Tuecke. Nexus: Runtime support for task-parallel programming languages. Technical Memo ANL/MCS-TM-205, Argonne National Laboratory, 1995.

[11] Ian Foster, Carl Kesselman, and Steven Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, 25 August 1996.

[12] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

[13] IEEE. *802.3 Carrier Sense Mulitple Access with Collision Detection (CSMA/CD)*. IEEEP, 1985.

[14] Myricom Inc. The gm message passing system. http://www.myri.com/scs/GM/doc/gm.pdf, July 2000.

[15] L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An interoperable framework for parallel programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.

[16] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel programming with message-driven objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[17] N. T. Karonis, B. Toonen, and I. Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–563, May 2003.

[18] Mark Nelson, William Humphrey, Attila Gursoy, Andrew Dalke, Laxmikant Kale, Robert D. Skeel, and Klaus Schulten. NAMD—a parallel, object-oriented molecular dynamics program. *Intl. J. Supercomput. Applics. High Performance Computing*, 10(4):251–268, Winter 1996.

[19] Scott Pakin and Avneesh Pant. VMI 2.0: A dynamically reconfigurable messaging layer for availability, usability, and management. In *The 8th International Symposium on High Performance Computer Architecture (HPCA-8), Workshop on Novel Uses of System Area Networks (SAN-1)*, Cambridge, Massachusetts, February 2002.

[20] Pradeep Kumar Panjwani. Monitoring and compression framework in virtual machine interface 2.0. Master's thesis, University of Illinois at Urbana-Champaign, 2002.

[21] Avneesh Pant, Sudha Krishnamurthy, Rob Pennington, Mike Showerman, and Qian Liu. VMI: An efficient messaging library for heterogeneous cluster communication. http://www.ncsa.uiuc.edu/Divisions/CC/ntcluster/VMI/hpdc.pdf, 2000.

[22] Gregory F. Pfister. An introduction to the infiniband architecture. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. IEEE/Wiley Press, New York, 2001.

[23] J. Postel. User datagram protocol. RFC 768, August 1980.

[24] J. Postel. Internet protocol. RFC 791, September 1981.

[25] J. Postel. Transmission control protocol. RFC 793, September 1981.

[26] Bjarne Stroustrup. The C++ programming language (third edition). *Addison Wesley, ISBN 0-201-88954-4*, 1997.

[27] T. H. von Eicken. *Active Messages: an Efficient Communication Architecture for Multiprocessors.* Ph.D. thesis, Computer Science, Graduate Division, University of California, Berkeley, CA, 1993.