

JMPI: IMPLEMENTING THE MESSAGE PASSING INTERFACE STANDARD IN
JAVA

A Thesis Presented

by

STEVEN RAYMOND MORIN

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

September 2000

Electrical and Computer Engineering

© Copyright by Steven Raymond Morin 2000

All Rights Reserved

JMPI: IMPLEMENTING THE MESSAGE PASSING INTERFACE STANDARD IN
JAVA

A Thesis Presented

by

STEVEN RAYMOND MORIN

Approved as to style and content by:

Israel Koren, Chair

C. Mani Krishna, Member

Ian Harris, Member

Seshu B. Desu, Department Head
Electrical and Computer Engineering

ACKNOWLEDGMENTS

I would like to thank Israel Koren and C. Mani Krishna for their generous patience, guidance and support throughout my graduate work and studies at the University of Massachusetts. I would also like to thank Professor Ian Harris for serving as a member on my thesis committee.

In addition, my graduate work would not have been possible without the generous support, and patience of Mark Wingertsman, Ben Marsden, and the entire staff of Engineering Computer Services. Mark and Ben have gone out of their way for me on numerous occasions and for this I am eternally grateful. I would also like to extend my gratitude to Stephen Cook and the entire staff of the Computer Science Computing Facility, especially with their patience with my work schedule during the Spring 2000 semester!

Furthermore, this work would not have been possible without the support and friendship of my lab mates: Joshua Haines, Vijay Lakamaraju, Siva Murugesan, Osman Unsal, Luis Villalta and Eric Ciocca. A special thanks goes to Vijay for sharing his love of crossword puzzles over lunch!

I would like to thank my parents and family for their love and patience during my graduate work. Finally, I would like to thank Kimberly West for her love, and patience and support.

ABSTRACT

JMPI: IMPLEMENTING THE MESSAGE PASSING INTERFACE STANDARD IN JAVA

SEPTEMBER 2000

STEVEN RAYMOND MORIN

B.S., UNIVERSITY OF MASSACHUSETTS AMHERST

M.S.E.C.E, UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Israel Koren

Over the past decade, researchers have investigated ways to leverage the power of inexpensive networked workstations for parallel applications. The Message Passing Interface (MPI) standard provides a common API for the development of parallel applications regardless of the type of multiprocessor system used. Recently, the Java programming language has made significant inroads as the programming language of choice for the development of Internet applications. We propose that the Message Passing Interface standard and Java are complementary technologies and develop a reference implementation written completely in Java. In this thesis, we discuss the results of our port, including adherence to emerging Java MPI standards and performance results.

TABLE OF CONTENTS

	Page
ACKNOWLEDGMENTS	iv
ABSTRACT	v
LIST OF TABLES	viii
LIST OF FIGURES	ix
CHAPTER	
1. INTRODUCTION TO MESSAGE PASSING ARCHITECTURES	1
1.1 Introduction	1
1.2 Data Parallelism.....	1
1.3 Distributed Computing	2
1.4 Message Passing Architectures	3
1.5 Parallel Virtual Machine	5
1.6 Message Passing Interface	6
2. INTRODUCTION TO THE JAVA PROGRAMMING LANGUAGE.....	8
2.1 The JAVA Programming Language	8
2.2 Simplicity	8
2.3 Features of Java.....	9
2.4 Security.....	10
2.5 Performance	11
2.6 Object Serialization	11
2.7 Remote Method Invocation (RMI)	12
3. RELATED RESEARCH.....	13
3.1 Introduction	13
3.2 JavaMPI.....	13
3.3 mpiJava	15
3.4 Distributed Object Group Metacomputing Architecture	17
4. CONSIDERATIONS FOR PORTING THE MPI STANDARD TO JAVA	20
4.1 Java's Inherent Qualities for Parallel Programming	20
4.2 Message Passing in JMPI.....	21
4.3 JMPI Architecture	24

4.4	Java MPI Application Programmer's Interface.....	25
4.5	Important API Considerations in Java.....	27
4.5.1	Internal Objects	27
4.5.2	Restrictions on Derived Datatypes	28
4.5.3	Multi-Dimensional Arrays and Offsets.....	28
4.5.4	Error Handling.....	30
4.6	Communications Layer.....	31
5.	TESTING METHODOLOGY	35
5.1	Introduction to MPI Standards Adherence.....	35
5.2	IBM Test Suite.....	36
5.3	Testing Methodology	37
5.4	JMPI Test Results	37
6.	JMPI PERFORMANCE.....	41
6.1	Performance	41
6.2	Test Environment	42
6.3	PingPong.....	43
6.3	Mandelbrot Curve	47
6.4	Nozzle.....	52
6.5	Conclusion	55
7.	CONCLUSIONS AND FUTURE RESEARCH	57
7.1	Conclusions and Future Research	57
	APPENDIX: GETTING STARTED WITH JMPI.....	58
A.1	Introduction.....	58
A.2	Downloading the JMPI software.....	58
A.3	Downloading the Java Development Toolkit.....	59
A.4	Configuring the MPI runtime environment	60
A.5	Creating the machinefile.....	61
A.6	SendMessage.java : a sample Java MPI Application.....	62
A.7	Compiling SendMessage	62
A.8	Running SendMessage	62
	BIBLIOGRAPHY.....	65

LIST OF TABLES

Table	Page
1. Test results of NPAC test suite against JMPI.....	38

LIST OF FIGURES

Figure	Page
1. Four process Distributed Parallel Application	22
2. JMPI Architecture	24
3. MPI Class Organization.....	26
4. PingPong performance between two nodes over fast Ethernet.....	44
5. PingPong Performance on a dual-processor system.....	45
6. PingPong performance on a single processor Linux system.	46
7. Output of the Mandelbrot curve application.....	49
8. Performance of the Mandelbrot application over fast Ethernet.	51
9. Performance of the Mandelbrot application within a dual-processor system.....	52
10. Performance of the Mandelbrot application within a single processor system.....	53
11. A screenshot of the Nozzle application.	56
12. Example virtual machine configuration file.....	61
13. SendMessages.java Example Java MPI Application	64

CHAPTER 1

INTRODUCTION TO MESSAGE PASSING ARCHITECTURES

1.1 Introduction

In an effort to solve problems too complex with theoretical approaches and too dangerous or expensive with empirical approaches, scientists turn to simulation models for solving these problems. Some problems, such as global climate modeling demand more computational resources than a single processor machine can provide. With the cost of parallel computers outside the reach of most budgets, researchers instead form parallel supercomputers out of existing in-house workstations connected via a network. Parallel applications are developed with message passing libraries, freeing developers from the cumbersome task of network programming, and allowing developers to concentrate on their algorithms.

1.2 Data Parallelism

A parallel computer consists of two or more identifiable, identical processors working together in parallel to solve a problem. Parallel computers are used to speed up algorithms in which a single operation is simultaneously applied to all elements of a data structure, such as an array. One example of an algorithm with inherent data parallelism is matrix multiplication. The product of two $m \times m$ matrices is computed with an array $m \times m$ processors, where processor P_{ij} is responsible for computing the dot product of the i th row and j th column. Each processor executes the same sequence of instructions on different elements of the matrix array. The algorithm completes after m iterations. The

earliest parallel computers consisted of vector processors, such as the Thinking CM-2. The vector model was later extended to include parallel computers with processors that operate on collections of data structures.

Shared memory multiprocessors have a single large contiguous address space accessible to all processors. Memory access by individual processors to the shared address space is coordinated with implicit hardware locks or explicit software locks defined within the parallel algorithm. Communication between processors is done through the shared address space. In contrast, distributed memory multiprocessors have a separate cache and address space for each processor. Communication between processors is done with explicit remote memory operations. With remote memory operations, one processor is able to explicitly access the address space of another processor, using the **get** operator to retrieve a word from remote memory and the **put** operator to store a word in remote memory. The cost of building a dedicated multiprocessor computer with hundreds or thousands of processors is too prohibitive for most researchers to use with their applications.

1.3 Distributed Computing

A distributed computer is formed when a set of independent workstations that are connected to each other through a network such as Ethernet, are used to solve a single problem. The combined resources of the general-purpose workstations connected can often exceed those of a dedicated single high-performance computer at a fraction of the cost. An example of a distributed computing application is a brute force attack against

the ciphertext of a plaintext message encrypted with the Digital Encryption Standard (DES). The ciphertext is distributed to thousands of workstations over the Internet. Each workstation is assigned to search through a small subset of the keyspace, and reports the results of the local search back to the distribution host. The most recent DES challenge, which distributed the brute force attack over thousands of different workstations, produced the correct 56-bit decryption key in just under 24 hours.

1.4 Message Passing Architectures

A message passing architecture defines communication primitives that parallel processes use to communicate and synchronize with other parallel processes. Primitives include communication functions such as send, receive, and broadcast, and synchronization primitives such as barrier. One major benefit of using message passing is that the primitives provide an abstraction of how the underlying hardware is organized. On shared-memory multiprocessors, the communication primitives use shared memory to transfer data between processes. On distributed-memory multiprocessors, the communication primitives use remote memory get and put operators to transfer data between processes. When a standard message-passing library is used, porting an application to another multiprocessor architecture normally requires just a recompile.

The second major benefit of using a message passing communication architecture is that a virtual parallel computer can be formed using inexpensive workstations connected via a network. The message-passing library is responsible for maintaining the state of the virtual parallel computer, distributing processes of the parallel application to individual

workstations and for providing reliable communication and synchronization between distributed processes. The virtual parallel machine is homogeneous when all workstations in the virtual parallel machine are of the same architecture and run the same operating system. The virtual parallel machine is heterogeneous when the workstations in the virtual parallel machine are of different architectures and operating systems. Hosts within a heterogeneous virtual parallel machine may internally represent primitive data types differently. For example, a long integer on an Alpha processor is represented using eight bytes versus four bytes for a short integer on an Intel Pentium Pro Processor. Furthermore, although two processors may use the same number of bits to represent the primitive type, one processor may internally represent the primitive type in little endian format, while another represents the primitive in big endian format.

In this case, the message passing library is required to convert the types to an architecture neutral format. The external data representation (XDR) format defines an architecture neutral representation of primitives. In a heterogeneous virtual machine, the message passing library automatically encodes primitives into XDR representation at the sender, and decodes from XDR into native representation at the receiver. The application developer is responsible for compiling a binary for each of the architectures included in the virtual parallel machine. Although the virtual parallel machine performs significantly slower than a dedicated multiprocessor, the cost of dedicated multiprocessors is often too expensive for most researchers. The cost of building a virtual parallel machine from in-house workstations is relatively low, and parallel applications can be run at night when most of the workstations are idle.

1.5 Parallel Virtual Machine

One of the first message passing libraries released for forming a virtual parallel machine is called Parallel Virtual Machine (PVM). First developed as a prototype in the summer of 1989 at the Oak Ridge National laboratory by Vaidy Sunderam and Al Geist, PVM 2.0 was developed at the University of Tennessee and released to the public in March, 1991. After receiving initial feedback from developers, a complete rewrite was undertaken, and PVM 3.0 was released in February 1993. PVM is available for public download, and has been ported to a large number of UNIX platforms including: Intel processors running Linux or FreeBSD, Digital Alpha processors running Digital UNIX, SPARC processors running Solaris, and Silicon Graphic workstations running IRIX. The PVM library exploits native messaging support on platforms where such support is available, such as distributed memory multiprocessors like the Intel Hypercube and shared memory multiprocessors like the SGI Challenge. Parallel Virtual Machine is still in widespread use; the latest official release is 3.4.

Parallel Virtual Machine is composed of two components: the PVM daemon and a communication library. The daemon runs on each workstation of the parallel virtual machine, and is responsible for maintaining the state of the parallel virtual machine, and providing some routing of messages between processes. The state of the virtual machine changes when workstations are added and removed. In addition to providing functions for communication, the library contains functions that allow applications to partition and decompose data, to add and remove tasks from the virtual machine and to add and

remove hosts from the virtual parallel machine. The library is available to applications written in C, C++ and Fortran. The configuration of the virtual machine and a current list of tasks running can be monitored with a library call from within the application, or from a graphical user interface.

1.6 Message Passing Interface

During the spring of 1992, the Center for Research in Parallel Computation sponsored a workshop on "Standards for Message Passing in Distributed Memory Environment". The workshop featured technical presentations by researchers and industry leaders on existing message passing implementations. Later that year, a group consisting of eighty members from forty organizations formed the MPI Forum whose charter was to define an open and portable message passing standard. The MPI Forum released the initial specification for the MPI-1 standard in August 1994. This standard defined a library of calls for point-to-point communication, collective group operations, process groups, communication domains, process topologies, environmental management and inquiry, profiling, and included Fortran and C bindings. The standard did not address implementation specific issues, which allowed implementers to exploit dedicated hardware found on machines with dedicated communication hardware, like the Intel Paragon. Twelve implementations of the MPI-1 standard are currently available, the two most respected implementations are MPICH developed at Argonne National Laboratory and Mississippi State University, and LAM developed at the Ohio Supercomputer Center. The members of the MPI Forum reconvened in the spring of 1995 to address ambiguities and errors in the MPI-1.0 specification, and released the MPI-1.1 standard in the summer

of 1995. The MPI Forum also reconvened in the summer of 1995 to address the broader issues left of the original standard due to time constraints. The MPI-2 standard was released in the summer of 1997 and included library functions for dynamic process management, input/output routines, one-sided operations and C++ bindings. There are no complete implementations of the MPI-2 standard as of the spring of 2000.

CHAPTER 2

INTRODUCTION TO THE JAVA PROGRAMMING LANGUAGE

2.1 The JAVA Programming Language

The JAVA programming language, developed by Patrick Naughton, Mike Sheridan and James Gosling of Sun Microsystems, was originally intended for controlling digital consumer devices, such as a cable television set top boxes and personal digital assistants. Java is designed to be simple, objected-oriented, architecture neutral, portable, dynamic, distributed, robust, multi-threaded language derived from C and C++. Although the JAVA initiative started in the summer of 1991, the language would remain obscure until the spring of 1995 when Netscape signed an agreement to license and incorporate the technology into the upcoming release of their World Wide Web Browser. Since then, the language has gained widespread acceptance from the academic and research communities, and from industry, with Intel, Microsoft, Novell, Oracle and other corporations signing licenses to incorporate Java technology into their core products. The first public Java Development Kit release occurred in January 1996 and it didn't take long for developers to release new products and applications using Java technology.

2.2 Simplicity

Java is an object-oriented language based on C and C++. The Java designers wanted to create a language that programmers could learn quickly, and removed some of the more complicated constructs from C and C++ that are notorious for creating problems for programmers. Examples of these constructs include: operator overloading, pointers,

multiple inheritance, gotos and union types. Operator overloading is redefining an operator such as the +, -, *, and / so that the operators call a method with a set of unary or binary parameters. Pointers, which are confusing and are the cause of a large amount of programmer errors, were replaced with reference types. Reference types contain a handle to a Java object rather than pointing to a physical location in memory. Multiple inheritance, where a class inherits methods and fields from two or more parents, was also removed. The **goto** statement was eliminated and replaced with a labeled **break** and **continue** statements. The preprocessor was removed and replaced with a package system. Structures were removed and replaced with classes and strongly typed primitives. The union structure was completely removed.

2.3 Features of Java

Java is an interpreted language and the Java compiler produces bytecode to execute on the Java Virtual Machine instead of producing executable code for a designated target platform. Java bytecode is architecture neutral and portable and the bytecode can be executed on any target platform where the Java Virtual Machine is available. Java is a strongly typed language, where the representation of all primitive data types are explicitly defined in the Java Language Specification, and the compiler requires explicit method declaration. The Java Virtual Machine is responsible for run-time array bound checking, run-time cast checking, exception handling and garbage collection.

The Java language is dynamic, distributed and multi-threaded. The Java Virtual Machine is able to dynamically load classes at runtime, from the local file system or from

a remote site over the Internet. The Java Native Interface specifies a mechanism that allows Java applications to access native code written in C, C++ or assembly. Introspection allows a Java class to dynamically probe for runtime information about loaded classes, such as available constructors, member functions, and fields. Remote Method Invocation defines a mechanism that allows Java applications to execute remote objects available over the network. The Java Language Specification defines support for multiple threads of execution, object synchronization, thread scheduling operations and thread groups in the base language.

2.4 Security

Java includes more security features than are found in traditional programming languages. The Java Virtual Machine performs byte-code verification to prevent illegal byte-code combinations, and performs run-time bounds checks on all array types. Virtual Machines integrated in Web Browsers execute applets in a “sandbox” model, where access to the local file system and network operations is severely restricted. The Java Language Specification doesn't include support for pointers, so a Java application can not access random areas of memory outside of the Java Virtual Machine. The reference type stores a handle to objects that are dynamically created. The handle is used by the Java Virtual Machine to reference fields or methods in main memory. Organizations and application developers can attach a digital signature to their applications. The digital signature is used to preserve the integrity of Java bytecode, preventing unauthorized modifications, and providing authenticity.

2.5 Performance

Interpreted languages perform worse than compiled languages and applications developed in Java are no exception. Java applications compiled with version 1.2 of the Java Development Kit execute approximately ten times slower than applications written in C. The Just-In-Time Compiler, available for some implementations of Java, compiles Java bytecode to native assembly language of the target machine transparently before execution. The Just-In-Time compiler vastly improves the performance of most Java applications comparable to a performance level enjoyed by most C applications.

2.6 Object Serialization

Data Marshalling is supported in Java through Object Serialization. Object Serialization is the process of encoding a Java object as an array of bytes. Similarly, Object Deserialization is the process of instantiating an object from an array of bytes. Object Serialization supports entire graphs of objects; for example, a single call can serialize the complete contents of a linked list. Objects can be saved into a file and re-opened at a later time, or transmitted over the network to another Java application via TCP/IP. Specific fields within an object can be marked as transient; these are fields that the Object Serializer is instructed to skip over because their context would be meaningless when deserialized. The Object Serialization libraries can be customized by overriding the `readObject()` and `writeObject()` methods. For example, custom serialization would allow a programmer to compact an array before serialization, and compute the correct number of elements after de-serialization. Object Versioning

prevents conflicts caused when a newer version of the same object is serialized by an older version of the same object.

2.7 Remote Method Invocation (RMI)

Remote Method Invocation (RMI) allows the Java programmer to offer the services of a local object remotely over the network. A programmer develops an interface for the object; the interface defines the signature of methods of the object available remotely. An implementation class is then coded based on this interface. The `rmic` compiler is used to generate linkage information for this class in the form of client stubs and server skeletons. The registry serves the role of Object Manager and Naming Service. When a server wishes to make a local object available to remote objects via RMI, it registers the interface and server skeleton with the registry. Before the client is able to invoke this method, it must contact the registry to locate the object. If the object has been registered, the registry returns a client stub to the client. This client stub provides information on how to marshal the parameters to the object. When the client invokes the object, the server skeleton code is used to marshal data between the registry and the server. Server objects can also be located via a special uniform resource locator (URL) of the form “`rmi://hostname[:port]/ObjectName`”. If omitted from the URL, the default port is 1099.

CHAPTER 3

RELATED RESEARCH

3.1 Introduction

Over the last four years, three separate research groups have investigated approaches on how to port the Message Passing Interface bindings to Java: JavaMPI by Sava Mintchev at the University of Westminster, mpiJava from the Northeast Parallel Architectures Centre at Syracuse University, and Distributed Object Group Metacomputing Architecture (DOGMA) by Glenn Judd at Brigham Young University. JavaMPI and mpiJava are wrapper-based implementations. For each MPI function, an analogous wrapper method is written in Java. This wrapper maps the Java parameters into C datatypes, invokes the equivalent C function, and finally maps the results back into Java objects. MPICH and LAM are often used in wrapper-based implementations. The DOGMA provides a subset of the MPI bindings natively in Java.

3.2 JavaMPI

JavaMPI was the first attempt to provide a Java binding for the MPI-1 standard using JDK 1.0.2. JavaMPI was developed at the University of Westminster by Sava Mintchev and released in the fall of 1997. JavaMPI is a set of wrapper functions which provide access to an existing native MPI implementation such as MPICH and LAM using the Native Method Interface provided with JDK 1.0.2. The Native Method Interface (NMI) allows Java programmers to access functions and libraries developed in another programming language, such as C or Fortran. The wrapper functions were generated

with JCI, a tool to automate the development of wrapper functions to native methods from Java. The Java bindings provided nearly conform to the MPI-1 specification. NMI has since been replaced with the Java Native Interface starting with JDK 1.1.

JavaMPI consists of two classes, MPIconst and MPI. The MPIconst class contains the declarations of all MPI constants and MPI_Init() which initializes the MPI environment. The MPI class contains all other MPI wrapper functions. The MPI C bindings include several functions that require arguments to be passed by reference for the return of status information. Java doesn't support call by reference. Instead, the use of objects is required when calling an MPI function with a call by reference parameter. This complicates application development in two ways: objects need to be instantiated with the new() constructor before using in an MPI function call, and the value of the variable is accessed through a field, such as *rank.val*. Pointer arithmetic is not supported in Java, so elements of an array can not be passed to functions as arguments as with C or Fortran. The programmer can use a function from JCI *JCI.section(arr, i)* to pass the *i*th element of array *arr*. Java doesn't support passing a subrange of an array. If the programmer wishes to send a subrange of an array to another process, the application must first create an MPI derived datatype and then send the MPI derived datatype.

The bindings as provided by JavaMPI significantly alter the program structure of an MPI application. As a result, porting an existing MPI application requires significant structural modification of the source code.

3.3 mpiJava

mpiJava provides access to a native MPI implementation through the Java Native Interface. mpiJava was developed at the Northeast Parallel Architectures Center at Syracuse University and released in the fall of 1998. The approach taken in the mpiJava implementation was to define a binding that is natural to use in Java. mpiJava models the Java bindings as closely as possible to the C++ bindings as defined in the MPI-2 standard and supports the MPI-1.1 subset of the C++ bindings. The mpiJava class hierarchy is organized as the C++ class hierarchy is defined in the MPI-2 specification and consists of six major classes, MPI, Group, Comm, Datatype, Status and Request. Figure 1 shows the organization of the six classes. The MPI class is responsible for initialization and global constants.

The Comm class defines all of the MPI communication methods such as send and receive. The first argument to all communication methods specifies the message buffer to be sent or received. The C and C++ bindings expect the starting address of a physical location in memory, with one or more elements of an MPI primitive type. The Java bindings expect an object to be passed. This Java object is an array of one or more elements of primitive type. Because Java doesn't support pointer arithmetic, all communication methods defined in the Java bindings take an additional parameter offset, which is used to specify the starting element in the array.

Java doesn't support call by reference parameters to methods, the only viable method of returning status from a method call is via the return value. Different approaches on

handling multiple return values from methods as defined in the MPI-1.1 specification were taken depending on the method call. If an MPI method modified elements within an array, the count of modified elements is returned. When an MPI method returns an array completely modified by the method, no count of elements is returned. The number of elements modified is available from the **length** member of the array. When an object is returned with a flag indicating status, the Java bindings omit the flag and return a null object when the method was unsuccessful.

Future research is being conducted on the use of object serialization within the mpiJava Java bindings. Object Serialization is a feature within the Java language that allows the member variables of a class and its parent classes to be written out to a file or across a network connection. When the class contains member variables are other objects, the member variables of the member objects will be serialized as well. The state of the object is restored by deserializing the object. The C bindings define a mechanism that describes the layout in memory of a C structure as an MPI derived datatype. After the datatype has been defined, the programmer can later send the derived datatype to other processes. Object Serialization in Java eliminates the need of creating a derived datatype to transmit the member fields of an object to another MPI process. The programmer could simply pass the object to a communication routine, and let object serialization take care of the rest.

3.4 Distributed Object Group Metacomputing Architecture

The Distributed Object Group Metacomputing Architecture (DOGMA) provides a framework for the development and execution of parallel applications on clusters of workstations and supercomputers. DOGMA is a research platform in active development by Glenn Judd at Brigham Young University. The DOGMA runtime environment, written in Java, is platform independent and supports dynamic reconfiguration and remote management, decentralized dynamic class loading, fault detection and isolation, and web browser and screen-saver node participation.

The foundation of the DOGMA runtime environment is based on Java Remote Method Invocation. Remote Method Invocation allows Java objects in one virtual machine to interact with other Java objects in a separate virtual machine. Java applications using Remote Method Invocation are able to dynamically locate remote objects, communicate with remote objects, and distribute bytecode of remote objects.

The Distributed Java Machine forms the second layer of the DOGMA runtime environment. The responsibility of the Distributed Java Machine is to connect several Java Virtual Machines into a single distributed machine. The nodes forming the distributed machine are organized into two categories: families and clusters. Families are nodes that have similar architecture and configuration. Clusters of nodes are nodes that are located near each other physically. The DJMManager is a centralized daemon responsible for the entire distributed machine, processing reconfiguration requests and faults. A NodeManager application runs on each node in the distributed machine,

applications communicate requests to the distributed machine through the local NodeManager. System topology is dynamic; nodes in the virtual machine can be added or removed at any time by an administrator. Groups of families and clusters of nodes can be stored in a configuration, and the configuration can be dynamically added or removed at run-time by the configuration manager. Web-browsers may participate in the distributed machine by loading the NodeManager applet. Applications are served from local disk space, or distributed from a World Wide Web server.

The Application Programmers Interface (API) is the topmost layer of the DOGMA environment. Two API specifications are available for use with application development: Message Passing Interface and Distributed Object Groups. The MPI implementation is completely written in Java and is approximately 80% complete. The API is based on the C++ bindings as specified in the MPI-1.1 specification as much as possible. The dominant API for the DOGMA environment is the Distributed Object Groups (DOG) environment. DOG allows applications to work with large groups of objects. The DOG API allows applications to perform asynchronous method invocations on large groups of objects, to dynamically partition and reassemble data, and to define group members for peer communication.

DOGMA is an ongoing research project. Future enhancements to DOGMA will include the ability to reload a newer version of the object without shutting down the runtime environment. Research is being performed on efficient object migration for

optimal performance of an application, decentralized system management, dynamic data partitioning enhancements and fault tolerance.

CHAPTER 4

CONSIDERATIONS FOR PORTING THE MPI STANDARD TO JAVA

4.1 Java's Inherent Qualities for Parallel Programming

The potential of Java as a programming language used by researchers to develop parallel applications running over a network of inexpensive workstations is vast. Java includes numerous features and a rich set of libraries that make it a natural language to develop parallel and distributed applications.

The first reason that Java is a natural language for the development of distributed parallel applications is portability. Java source code compiles to an architecture neutral byte code format, which runs on any machine the Java Virtual Machine has been ported to. In addition, the Java Language Specification specifies exactly how the primitive data types are represented: in big-endian order and in terms of their size (in bits). This complete specification of all primitive types eliminates the problems related to data conversion within a heterogeneous virtual machine.

The second reason that Java is a natural language for the development of distributed parallel applications is language simplicity. The language eliminates complex constructs that were a source of headaches for programmers. An example of this is the elimination of pointers and pointer arithmetic. C programmers often spend a large amount of time debugging code where a pointer inadvertently points to an invalid region of memory. Java replaces the pointer with the reference type, and the strong type checking system

ensures that array bounds aren't exceeded at runtime, and the garbage collector frees memory used by objects when the application no longer has any reference to those objects.

The final reason that Java is a natural language for developing distributed parallel applications is that Java includes functionality to assist with the development of distributed applications. Object Serialization allows user-defined data structure to be transferred to remote processes transparently with one call. With Remote Method Invocation, a process can invoke a method on a remote machine with the same calling semantics as a local object. Object Serialization and Remote Method Invocation are important components of a message passing architecture developed in Java.

4.2 Message Passing in JMPI

Java Message Passing Interface (JMPI) is an implementation of the Message Passing Interface for distributed memory multi-processing in Java. Unlike wrapper based implementations of MPI, such as mpiJava, JMPI is completely written in Java and runs on any host that the Java Virtual Machine is supported on. In contrast, mpiJava relies on the Java Native Interface (JNI) to communicate with an implementation of MPI written in C. As a result, mpiJava is restricted to running on hosts where a native MPI implementation has been ported. Furthermore, JNI introduces platform specific dependencies, which further reduces the application's portability. Finally, a wrapper-based implementation is more difficult to install and run.

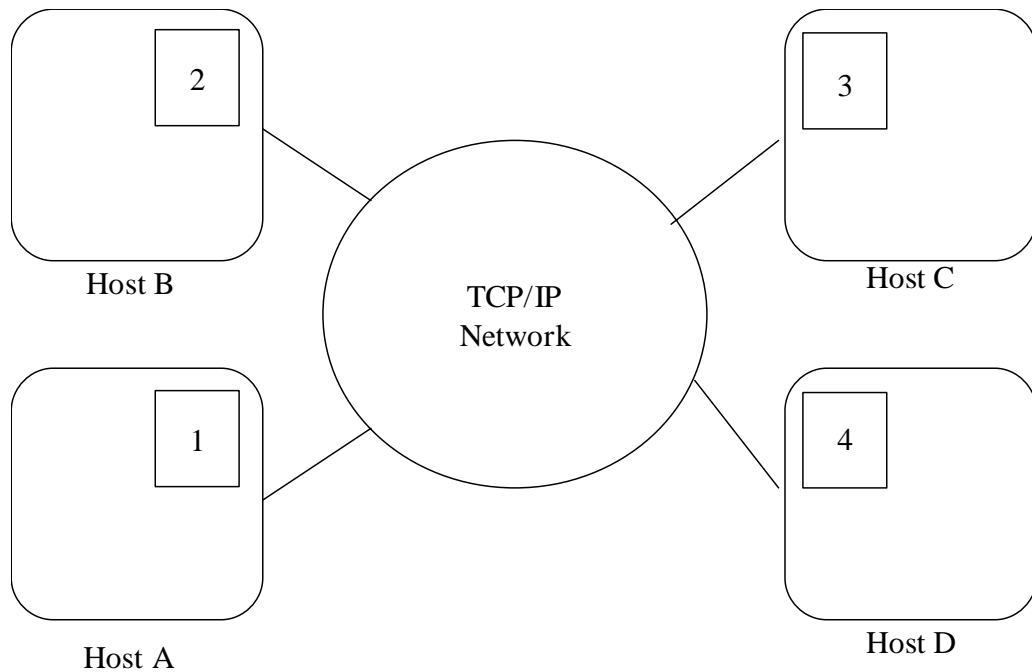


Figure 1. Four process Distributed Parallel Application

Figure 1 shows a four process parallel application running on a four processor distributed machine connected via a TCP/IP based network. The traditional model of message passing in this environment is via Berkeley Sockets. For example, lets assume that Host D has a message to send to Host A. At the time the distributed machine is formed, Host A opens a socket on a predetermined port and listens for incoming connections. Likewise, Host D creates a socket and connects to the open port on Host A. Once the connection has been established, messages can be sent in either direction between Hosts A or D. Messages sent between processes are sent as an array of bytes. As a result, the high-level data structures are encoded as byte arrays before the message is transmitted over the network.

JMPI implements Message Passing with Java's Remote Method Invocation (RMI). RMI is a native construct in Java that allows a method running on the local host to invoke a method on a remote host. One benefit of RMI is that a call to a remote method has the same semantics as a call to a local method. As a result, objects can be passed as parameters to and returned as results of remote method invocations. Object Serialization is the process of encoding a Java Object into a stream of bytes for transmission across the network. On the remote host, the object is reconstructed by deserializing the object from the byte array stream. If the object contains references to other objects, the complete graph is serialized.

For example, assume that Host B in Figure 1 wishes to send a message to Host C. At the time the distributed machine is formed, Host C registers a local object available for remote method invocation with a Java registry. The purpose of the registry is to function as a naming service. The registry allows a local method, such as the one running on Host B, to turn the object's Uniform Resource Locator (URL) into a reference to the remote method. Once the reference is bound, the process on Host B invokes the remote method with the Message as a single parameter. The method running on Host C inserts the message into a local First-In-First-Out (FIFO) queue, and notifies all blocked threads that a new message has arrived. The remote method returns a Boolean value of true to indicate the successful transmission of a message, otherwise false is returned.

4.3 JMPI Architecture

Figure 2 shows the architecture of JMPI. JMPI has three distinct layers: the Message Passing Interface API, the Communications Layer, and the Java Virtual Machine. The Message Passing Interface API, which is based on the proposed set of Java bindings by the Northeast Parallel Architecture Centre (NPAC) at Syracuse University, provides the core set of MPI functions that MPI applications are written to. The Communications Layer contains a core set of communication primitives used to implement the Message Passing Interface API. Finally, the Java Virtual Machine (JVM) compiles and executes Java Bytecode.

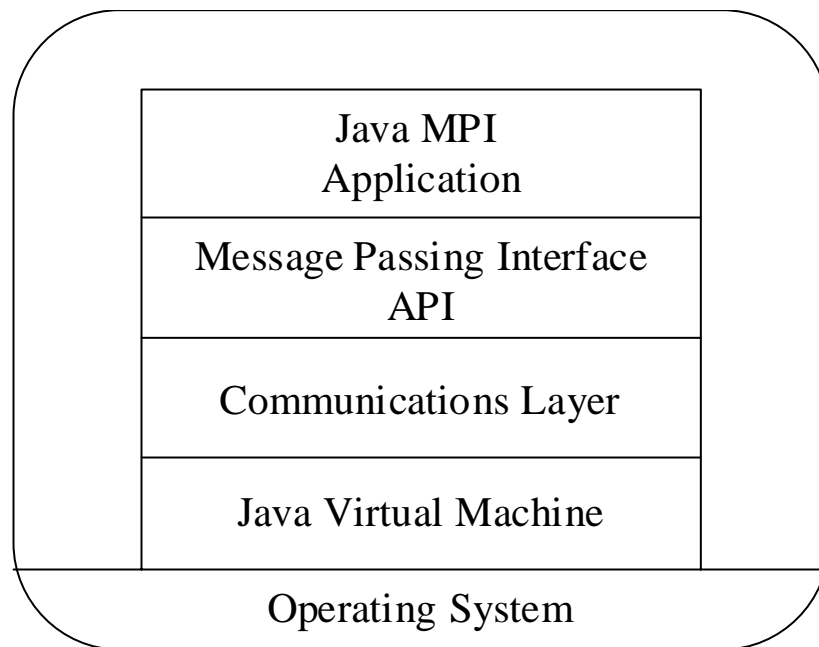


Figure 2. JMPI Architecture

4.4 Java MPI Application Programmer's Interface

The MPI Forum, a group consisting of eighty members from forty organizations, was formed in late 1992 to develop an open and portable message passing standard. The MPI-1 standard was released in August 1994, and defined a set of calls of point-to-point communication, collective group operations, process groups, communication domains, process topologies, environmental management and inquiry and profiling. Initially the standard included only a set of bindings for Fortran and C. Subsequently, the MPI Forum released the MPI-1.1 standard during the spring of 1995 to address ambiguities and errors in the MPI-1.0 standard, and to address broader issues previously not dealt with in the initial release. The MPI-2.0 standard was released in the summer of 1997 and included library functions for dynamic process management, input/output routines, one-sided operations and C++ bindings. Although portions of the MPI-2 standard have been retrofitted into MPI-1 implementations, a complete MPI-2 implementation has not been released.

Furthermore, the MPI Forum has not officially reconvened since the summer of 1997 when the MPI-2 standard was released. As a result, no official bindings exist for JAVA. The Northeast Parallel Architecture Centre has released a draft specification that closely follows the C++ bindings. Figure 3 shows the proposed organization of the MPI into six main classes: MPI, Group, Comm, Datatype, Status and Request. The Comm has two subclasses: the Intracomm class for intracommunicators, and the Intercomm class for intercommunicators. Furthermore, Intracomm is split into two subclasses: Cartcomm and Graphcomm.

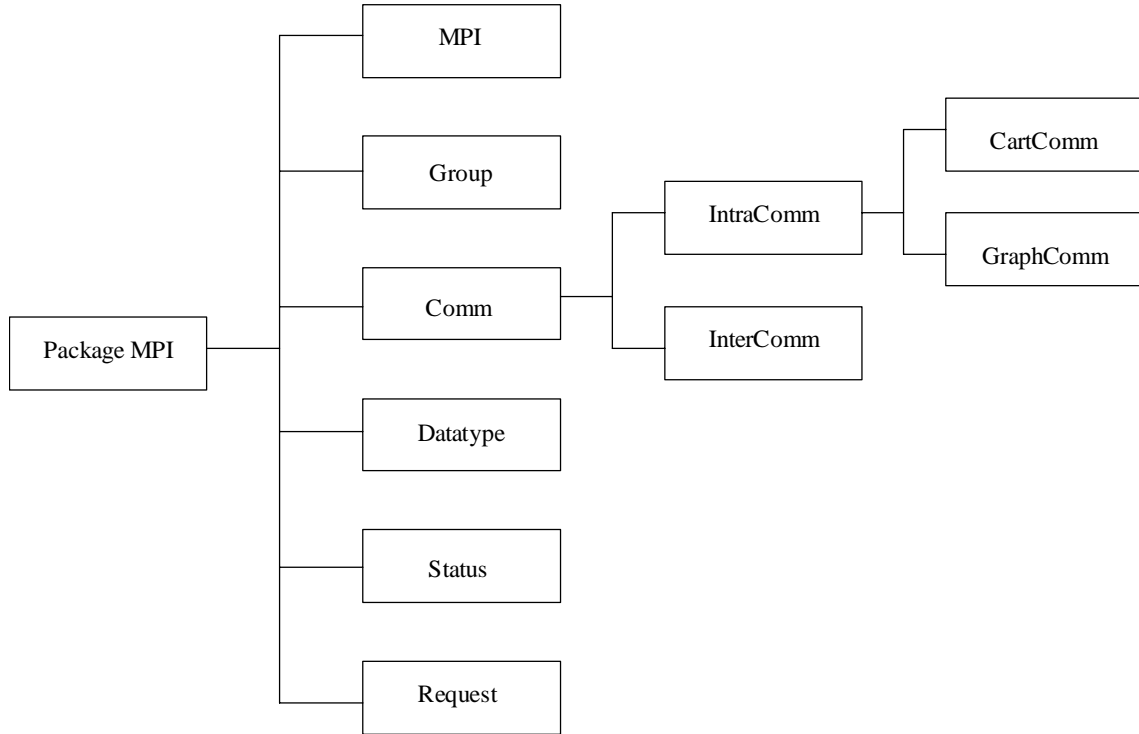


Figure 3. MPI Class Organization

Although no formal JAVA bindings exist yet for the MPI standard, we choose to follow the bindings as proposed by the Northeast Parallel Architectures Centre (NPAC). First of all, the proposed JAVA bindings are derived from and closely follow the official C++ bindings developed by the MPI Forum. Secondly, we felt it was important to maintain compatibility between our implementation of MPI and other implementations based on the proposed bindings. As a result, programmers don't need to modify their code for it to work with different JAVA implementations of MPI. Furthermore, we have been able to run the IBM Test Suite Validation software ported to Java by NPAC against our implementation without modifying source code.

4.5 Important API Considerations in Java

Although the MPI Bindings for Java closely follow the official MPI C++ bindings, several important modifications were made due to the design of Java. The following sections describe four of the more important changes in the proposed Java bindings from the official C++ bindings.

4.5.1 Internal Objects

The MPICH implementation of MPI is inherently object-oriented; core data structures, such as Communicators, Groups and Requests are *opaque* objects. An object is defined as opaque when its internal representation is not directly visible to the user. Instead, the program refers to an opaque object through the use of a handle. This handle is passed into and returned from most MPI calls. In contrast, the Java Bindings use objects to represent internal states: these objects are passed directly into and returned from most MPI methods. In addition, the Java Bindings define appropriate methods to work with these objects. For example, the IntraCommunicator Class defines send and receive methods which allow a process within the IntraCommunicator to send and receive messages from other processes within the IntraCommunicator. Similar to an opaque object, the internal representation of the Java object is not directly visible to the MPI application.

4.5.2 Restrictions on Derived Datatypes

The Java Virtual Machine does not support direct access to main memory, nor does it support the concept of a global linear address space. As a result, operations that map data structures to physical memory are not available to Java based MPI applications. Instead, the Java Bindings support Object Serialization for passing messages between processes. Object Serialization flattens the state of a Java object into a serial stream of bytes. An object is serializable when the programmer implements the `Serializable` interface in an object's declaration. Object Serialization is used whenever the `MPI.OBJECT` type is specified as an argument to any of the communication methods. Arrays of objects and strided arrays of objects can be serialized.

4.5.3 Multi-Dimensional Arrays and Offsets

Multi-Dimensional arrays in C and Fortran are mapped into equivalent one-dimensional arrays sequentially in memory. For example, the declaration of

```
int a[4][3];
```

is equivalent to

```
int b[12];
```

with the size of each array totaling 48 bytes. The first row of a, `a[0][0]`, `a[0][1]`, and `a[0][2]`, are mapped to `b[0]`, `b[1]`, `b[2]`, respectively, the second row of a, `a[1][0]`, `a[1][1]`, and `a[1][2]` are mapped to `b[3]`, `b[4]`, `b[5]`, respectively, and so on. Multi-dimensional arrays passed as buffers into MPI routines are interpreted as their one-dimensional counterparts with the same element type as the original multi-dimensional array.

Java represents the multi-dimensional arrays differently. In Java, an n dimensional array is represented as a one-dimensional array of (n - 1)-dimensional arrays. In the previous example, the `int[4][3]` in Java is equivalent to a four element array of a three element array of integers. This complicates an MPI implementation in Java because multi-dimensional arrays are not stored sequentially as in C. Furthermore, Java prohibits direct access to physical memory. This prevents the MPI application from specifying an offset other than zero of the first element in the message buffer.

The C and Fortran MPI bindings have an implicit offset parameter to all functions which have a message buffer as a parameter. The implicit offset is always 0, because the user can directly specify the starting address of the message buffer. To specify an offset of 10 into a one-dimensional array, an application written in C passes the address-of the tenth element directly as the parameter (`&b[10]`). Java does not support direct access to memory, so the Java MPI bindings add an additional parameter to methods with message buffers as parameters that allow MPI applications to specify the starting element within the array as an integer offset.

JMPI provides supports for multi-dimensional arrays as if the arrays were laid out sequentially in memory as in C. Support for multi-dimensional arrays of any dimension is accomplished through the use of Java's Introspection. Introspection allows the Java application to dynamically determine the type of any Object. When a message buffer is passed as a parameter to an MPI method, the algorithm uses Introspection to determine

the number of dimensions within the message buffer and the number of elements that are in each dimension of the array. After the number of elements in each dimension is calculated, the reference to the array containing the first element in the message buffer as specified in the offset parameter is determined. The algorithm sequentially steps through each reference in the multi-dimensional array until the number of elements as specified in the MPI application is exhausted or the last element of the array is reached. In contrast, mpiJava is limited to arrays of one dimension. Attempts to use multi-dimensional arrays with an offset larger than the size of the innermost array will result in undefined values sent to other processes.

4.5.4 Error Handling

Error handling in the Java Bindings differs significantly from the C and Fortran bindings. For example, most MPI functions in the C API return a single integer value to indicate the success or failure of the call.

In contrast, Java throws exceptions when an error occurs. For example, when a program tries to index an array outside of its declared bounds, the `java.lang.ArrayIndexOutOfBoundsException` exception is thrown. The program can choose to catch the exception or propagate the exception to the Java Virtual Machine (JVM). The purpose of this code is to perform some cleanup operation, such as prompting for another filename when a `FileNotFoundException` is thrown. If the exception propagates to the Java Virtual Machine, a stack trace is printed and the application terminates.

The Java MPI bindings chose to use Exceptions when an error condition occurs rather than returning an integer error-code. First of all, the exception error-handling facility is built into the language, and all the run-time libraries provided with Java use it. In addition, Java does not support pass-by-reference calling semantics, so it becomes extremely clumsy to return multiple error codes.

4.6 Communications Layer

The Communications Layer has three primary responsibilities: virtual machine initialization, routing messages between processes, and providing a core set of communications primitives to the Message Passing Interface API. The Communications Layer is multi-threaded and runs in a separate thread than the MPI process. This allows for the implementation of non-blocking and synchronous communication primitives in the MPI API. Messages are transmitted directly to their destination via Remote Method Invocation.

The Communications Layer performs three tasks during virtual machine initialization: start an instance of the Java registry, register an instance of the Communication Layer's server skeleton and client stub with the registry, and perform a barrier with all other processes that will participate in the virtual machine. The Registry serves as an object manager and naming service. Each process within the virtual machine registers an instance of the Communications Layer with the local registry. The registered instance of

the Communication Layer is addressable with a Uniform Resource Locator (URL) of the form:

`rmi://hostname.domainname.com:portno/Commx`

where *portno* represents the port number that the registry accepts connections on, and *x* represents the rank of the local process, which ranges from 0 to $n-1$, where n is the total number of processes in the virtual machine. One of the command line parameters passed to each process during initialization is the URL of the Rank 0 Communications Layer. After the Communications Layer has started its local registry and registered its instance of the local Communications Layer, the last step during initialization is to perform a barrier with all other processes. The purpose of this barrier is two fold: ensure that all processes have initialized properly, and receive a table containing the URLs of all the remote processes Communications Layers within the virtual machine. The barrier is formed when all processes have invoked the barrier method on the rank 0 process. For processes 1 to $n-1$, this means a remote method invocation on the rank 0 process. Before the remote method can be invoked, the process must bind a reference to the remote object through its uniform resource locator. Once the reference is bound, the remote method is invoked in the same manner as a local method would be. The barrier on the rank 0 process uses Java's notify and wait methods to implement the barrier.

Messages between processes are passed as a parameter to a remote method invocation on the destination process. The Message is a serializable object that consists of fields to represent the source rank, the destination rank, and message tag. In addition, the Message incorporates a Datatype object to indicate the type of data being sent in the

message, as well as the actual data. The remote method inserts the message object into the local processes First-In-First-Out (FIFO) Message Queue, and then notifies all locally blocked processes that a new message has arrived through Java's synchronization routines. For synchronous mode sends, where completion of the call indicates the receiver has progressed to the matching receive, the Communication Layer blocks and waits for notification from the local process that the matching receive has started. Java's wait and notify methods are used to implement synchronous mode sends. For all other communication modes, the remote method returns after the message has been inserted into the local message queue.

The local process receives messages by invoking a local method in the Communications Layer. This method checks the incoming message queue for a message that matches the source rank and message tag passed as parameters. If a matching message is found, the receive call returns with this message. Otherwise, the call blocks and waits for a new message to come in. The message queue is implemented with a Java Vector Class for two reasons: synchronized access allows more than one thread to insert or remove messages at a time, and Vectors perform significantly faster than a hand-coded linked list implementation. In addition, Vectors also support out-of-order removal of messages, which is required to implement the MPI receive calls.

The Communications Layer provides communication primitives from which all other MPI bindings are implemented. The blocking point-to-point primitives have been discussed in the previous paragraphs. The non-blocking versions of the point-to-point

primitives are implemented by creating a separate thread of execution, which simply calls the respective blocking version of the communication primitive. Collective communication primitives, such as a broadcast to all processes are built on top of the point-to-point primitives. A separate message queue is used to buffer incoming collective operations to satisfy the MPI specification document. Two probe functions allow the message queue to be searched for matching messages without retrieving them. Finally, a barrier primitive allows all processes within an intracommunicator to synchronize their execution.

CHAPTER 5

TESTING METHODOLOGY

5.1 Introduction to MPI Standards Adherence

Although no formal MPI standard yet exists for the Java programming language, the Northeast Parallel Architecture Centre (NPAC) at Syracuse University has proposed a draft standard. This draft standard proposes a set of Java language bindings so that MPI implementers and Java application programmers can take advantage of the power of MPI from within Java. Although the draft standard is based on the official C++ bindings as adopted by the MPI Forum in the MPI-2 specification, modifications to this standard have been made to accommodate the language differences between C++ and Java. A short list of these differences include: the introduction of the MPI.OBJECT datatype, restrictions on the use of the struct derived datatype, an offset parameter to all communication primitives to indicate the start of a message buffer, and the use of Java exceptions to report error conditions.

Although this document is not an official MPI Forum standard, JMPI's adherence to this proposed standard is important for one reason: compatibility. Adherence to the proposed standard ensures that behavior of programs that are written to this specification does not change when the program is ported to other wrapper-based or native implementations. Additionally, other similar efforts are underway to implement MPI bindings for Java based on this draft standard.

5.2 IBM Test Suite

During the spring of 1993, IBM released a suite of C applications that test the functionality of an MPI implementation. The applications are organized into six main categories that test specific functionality within the MPI implementation: point-to-point communication operators, collective operators, context, environment and topology. NPAC has ported the IBM test suite from C to Java and released the Java test suite as part of their mpiJava wrapper based implementation. The functionality of the Java test suite is broken into seven main categories: collective communication operators, communicators, datatypes, environmental operators, group operators, point-to-point communication operators and topography operators. The NPAC test suite includes several new applications to test the Java specific extensions of a Java based MPI implementation, including several tests for proper handling of the MPI.OBJECT datatype.

The architecture of each test application included in the test suite is similar, so we will examine one test application: `sendrecv.java`. This application tests an MPI implementation of the `Comm.Sendrecv()` binding. `Sendrecv` is a point-to-point communication primitive that sends a message and receives a message from a remote process with one call. The test requires two processes: a master process and a slave process. This specific test determines if an array of integers of 1,000 elements is correctly passed between two MPI processes and checks to make sure that the buffer used to send data is the same buffer used to receive it. The master process will report an error if the master detects incorrect data or a message from a process other than its slave.

5.3 Testing Methodology

In addition to the normal testing that accompanies every software development process, JMPI was tested against the NPAC test suite to ensure the correctness of the MPI implementation. This test environment consisted of two machines: a dual-processor Pentium-III running Solaris X86 and a single-processor running RedHat Linux 6.1. The two test machines were connected via a Fast Ethernet Network. Each test was run with three different test environments. In the first test environment, both processes of the test were distributed on the dual processor machine. Similarly, the second test environment distributed both processes of the test on the single processor machines. Finally, the third test environment distributed each process on its own machine and communication occurred over the fast Ethernet channel.

5.4 JMPI Test Results

The NPAC test suite was used during the development of JMPI to test correctness of the implementation as functionality was added. In some cases, errors were reported as a result of misinterpretations of the draft standard. One example of this includes not setting Request objects to null upon successful completion of a non-blocking receive communication primitive. This feedback was an invaluable part of the development phase. Table 1 shows the final test results for JMPI. Pass indicates the successful completion of functionality tested without errors. Not Implemented indicates the functionality tested is not yet implemented in JMPI.

Category	Test	Status
Collective Communications	allgather	Pass
Collective Communications	allgatherO	Pass
Collective Communications	allgatherv	Pass
Collective Communications	allgathervO	Pass
Collective Communications	allreduce	Not Implemented
Collective Communications	allreduce_maxminloc	Not Implemented
Collective Communications	allreduceO	Not Implemented
Collective Communications	alltoall	Pass
Collective Communications	alltoallO	Pass
Collective Communications	alltoallv	Pass
Collective Communications	alltoallvO	Pass
Collective Communications	barrier	Pass
Collective Communications	bcast	Pass
Collective Communications	bcastO	Pass
Collective Communications	gather	Pass
Collective Communications	gatherO	Pass
Collective Communications	gatherv	Pass
Collective Communications	gathervO	Pass
Collective Communications	reduce	Not Implemented
Collective Communications	reduce_scatter	Not Implemented
Collective Communications	reduce_scatterO	Not Implemented
Collective Communications	reduceO	Not Implemented
Collective Communications	scan	Not Implemented
Collective Communications	scanO	Not Implemented
Collective Communications	scatter	Pass
Collective Communications	scatterO	Pass
Collective Communications	scatterv	Pass
Collective Communications	scattervO	Pass
Communicators	attr	Pass
Communicators	commdup	Pass
Communicators	compare	Pass
Communicators	intercomm	Not Implemented
Communicators	split	Pass
Datatype	hvec	Not Implemented
Datatype	hvecO	Not Implemented
Datatype	lbuf	Not Implemented
Datatype	lbuf2	Not Implemented

Table 1. Test results of NPAC test suite against JMPI.

Continued, next page.

Category	Test	Status
Datatype	lbubO	Not Implemented
Datatype	pack	Pass
Datatype	packO	Pass
Datatype	type_size	Pass
Datatype	zero1	Not Implemented
Datatype	zero1	Not Implemented
Environment	abort	Pass
Environment	initialized	Pass
Environment	procname	Pass
Environment	wtime	Pass
Group	group	Pass
Group	range	Pass
Point-to-Point	bsend	Pass
Point-to-Point	bsend_free	Pass
Point-to-Point	bsend_org	Pass
Point-to-Point	bsend_test	Pass
Point-to-Point	bsendO	Pass
Point-to-Point	buffer	Pass
Point-to-Point	getcount	Pass
Point-to-Point	getcountO	Pass
Point-to-Point	interf	Pass
Point-to-Point	iprobe	Pass
Point-to-Point	isend	Pass
Point-to-Point	isendO	Pass
Point-to-Point	probe	Pass
Point-to-Point	rsend	Pass
Point-to-Point	rsend2	Pass
Point-to-Point	rsend2O	Pass
Point-to-Point	rsendO	Pass
Point-to-Point	sendrecv	Pass
Point-to-Point	sendrecv_rep	Pass
Point-to-Point	sendrecvO	Pass
Point-to-Point	seq	Pass
Point-to-Point	ssend	Pass
Point-to-Point	ssendO	Pass
Point-to-Point	start	Pass
Point-to-Point	startall	Pass
Point-to-Point	StartO	Pass
Point-to-Point	test1	Pass
Point-to-Point	test1O	Pass

Table 1 continued

Category	Test	Status
Point-to-Point	test2	Pass
Point-to-Point	test2O	Pass
Point-to-Point	test3	Pass
Point-to-Point	test3O	Pass
Point-to-Point	Testall	Pass
Point-to-Point	TestallO	Pass
Point-to-Point	Testany	Pass
Point-to-Point	TestanyO	Pass
Point-to-Point	Testsome	Pass
Point-to-Point	TestsomeO	Pass
Point-to-Point	Waitall	Pass
Point-to-Point	WaitallO	Pass
Point-to-Point	Waitany	Pass
Point-to-Point	WaitanyO	Pass
Point-to-Point	Waitnull	Pass
Point-to-Point	Waitsome	Pass
Point-to-Point	WaitsomeO	Pass
Point-to-Point	Wildcard	Pass
Topography	Cart	Not Implemented
Topography	Dimscrate	Not Implemented
Topography	Graph	Not Implemented
Topography	Map	Not Implemented
Topography	Sub	Not Implemented

Table 1 continued

CHAPTER 6

JMPI PERFORMANCE

6.1 Performance

One of the most important characteristics of a parallel programming environment is performance. Scientists won't invest the time to develop parallel applications in Java if the performance is significantly slower than the same application coded in C. Furthermore, Java starts with a slight disadvantage because the language is interpreted rather than executed. In this section, the performance of JMPI is compared to the performance of two other MPI implementations: MPICH and mpiJava. MPICH, from the Argonne National Laboratory, is an implementation of MPI completely written in C. mpiJava from Syracuse University is a wrapper based implementation of MPI for Java which uses MPICH as the native implementation of MPI. Two applications are used to compare performance of JMPI to other implementations: PingPong and Mandelbrot. PingPong measures the round-trip time required to transmit a message between two MPI processes. Mandelbrot calculates the Mandelbrot fractal on a 512 by 512 grid by distributing work to slave processes. The performance of the native MPICH implementation is measured with the application coded in C and the performance of mpiJava and JMPI is measured with the application coded in Java. A third application, Nozzle, shows the limitations of using a wrapper based implementation of MPI. Nozzle is a Java applet that simulates the flow of a non-viscous fluid through an axisymmetric (cylindrical) nozzle.

A comparison of performance JMPI to MPICH and mpiJava shows mixed results: performance of an application distributed over two workstations on an Ethernet network suffers significantly when small message sizes are used due to the extra overhead of Object Serialization and the added latency of the remote method invocation. This overhead becomes less significant as the size of the message passed increases. However, some applications terminated abnormally in mpiJava with the error of mis-matched message from the MPICH backend. Although the performance of JMPI lags behind mpiJava, applications are more stable in JMPI.

6.2 Test Environment

The test environment consists of two nodes connected via a fast Ethernet network. The first node contains two Intel Pentium-III processors each running at 450-MHz. The dual-processor is equipped with 256 MB of main memory, a fast Ethernet adapter and runs Solaris X86. The second node is a Dell Latitude laptop with a single processor running at 400 MHz. The laptop is equipped with 128 MB of main memory, a Cardbus based fast Ethernet adapter and runs RedHat Linux 6.1. The dual-processor has the Solaris production release of the Java Development Toolkit (JDK) for Solaris X86, version 1.2.2_05, while the laptop runs the freeware release of Blackdown JDK for Linux, version 1.2.2rc4. The core of the network is an eight-port NetGear fast ethernet switch. The performance of each application is measured in three ways: processes distributed on both nodes, processes distributed only on the dual-processor and processes distributed only on the single-processor.

6.3 PingPong

PingPong measures the round-trip delay of a message between two MPI processes and calculates the throughput available. The message consists of a one-dimensional array of bytes where the number of elements in the array varies as a function of 2^x , where x varies from 0 to 20. The smallest message is an array containing 1 element (1 byte) and the largest message is an array containing 1,048,576 elements (1 Megabyte). All elements in the array are initialized to a value as defined by the equation

$$A[i] = (\text{byte})'0'; //(double) 1. / (i + 1);$$

before the message is sent. Each message is sent a total of 64 times between the rank 0 and rank 1 process. The rank 0 process records the round-trip time of each message and produces an average round-trip time and average throughput, measured in Mb/sec, for each array.

Figure 4 shows that for small message sizes, the performance of JMPI is significantly slower than MPICH and mpiJava. MPICH uses the native C socket interface to transmit messages between processes while mpiJava uses the Java Native Interface (JNI) to marshal Java datatypes to MPICH. In contrast, JMPI uses Remote Method Invocation to transmit messages between the source process and the destination process. The poor performance for small message sizes is due to the overhead attributed to Remote Method Invocation. The RMI overhead consists of two parts: object serialization and the latency added by invoking a remote method. Object Serialization flattens Java Objects into an array of bytes for transmission across the network. This process of flattening increases the message size by a fixed constant of 325 bytes. For small message sizes, object

serialization adds a significant amount of overhead. To measure the amount of latency added by Remote Method Invocation, we have coded a simple remote method that performs no computation other than returning control of execution back to the caller. This method was invoked for 1,000 iterations and the average added latency of a remote method over an Ethernet network was 0.94 ms.

PingPong Performance between two nodes over fast ethernet

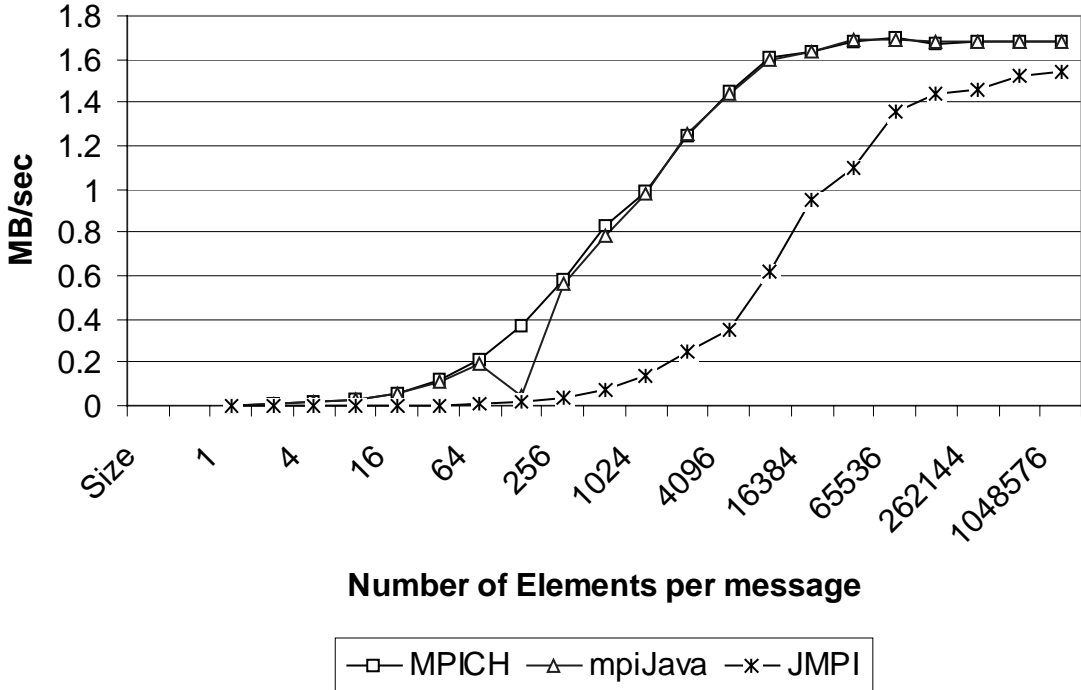


Figure 4. PingPong performance between two nodes over fast Ethernet.

Figure 5 shows the performance of the PingPong application on a dual processor Solaris X86 system. The poor performance of JMPI is attributed directly to the overhead of remote method invocation compared with the shared memory transport of MPICH. At the present time, JMPI processes that are started on the same machine are invoked in

separate java virtual machine environments and the processes use remote method invocation for message transport. Significant performance enhancements could be made if these processes were started as separate threads within the same virtual machine environment and the message transport between processes was a local method call within the virtual machine. Although two or more processes would execute in a single Java Virtual Machine, the processes would remain separately schedulable and run on separate processors when run within a Java Virtual Machine that implements native threading.

PingPong Performance on dual processor

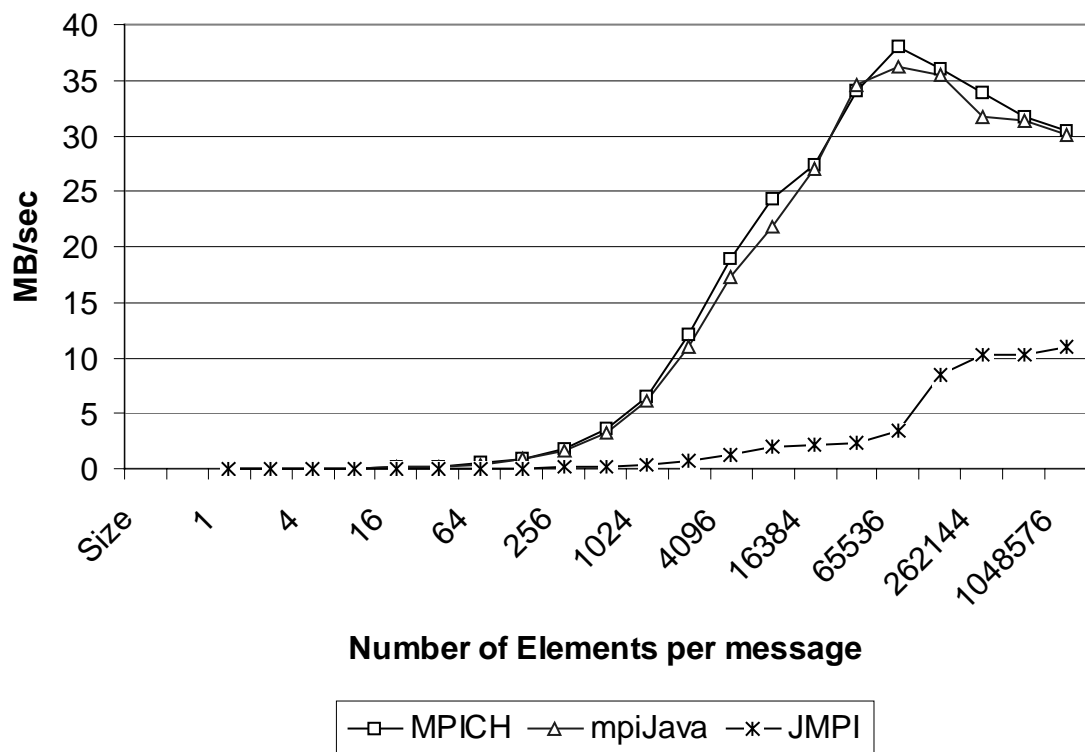


Figure 5. PingPong Performance on a dual-processor system.

PingPong Performance on single processor

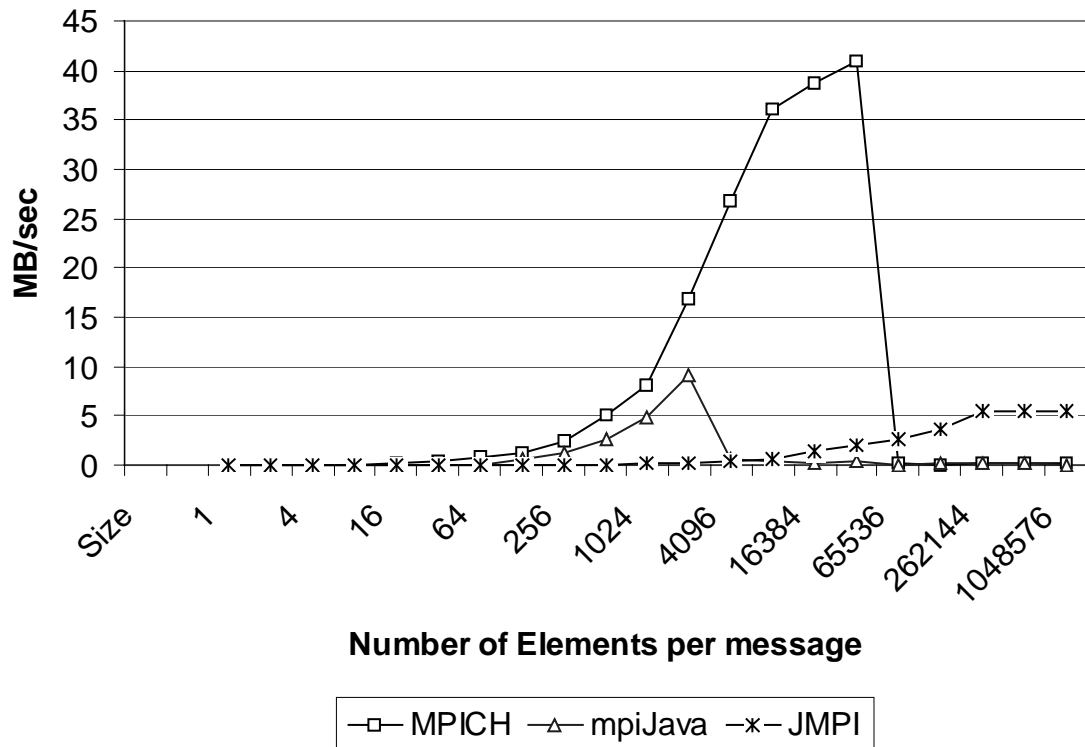


Figure 6. PingPong performance on a single processor Linux system.

Figure 6 shows that performance of PingPong on a single processor Linux system is significantly better than MPICH for large message sizes but continues to be lower for small message sizes. The MPICH implementation on Linux uses sockets rather than shared memory for message transport between processes on the same machine. This results in a bottleneck for message sizes greater than 32,768 elements for native C and 2,048 elements for mpiJava due to a lack of buffer space and the change in transport protocol used to exchange messages. For small message sizes, MPICH uses the eager protocol that assumes the remote process has sufficient memory to buffer messages.

However, for large messages MPICH transfers small sections of the message to the remote process and waits for an acknowledgement before sending the next section. Like the dual-processor case, significant performance enhancements could be made if these processes were started as separate threads within the same virtual machine environment and the message transport between processes was a local method call within the virtual machine.

6.3 Mandelbrot Curve

Perhaps one of the most intriguing facets of computer graphics is fractals. A fractal is a curve or surface that exhibits some degree of self-similarity. Fractals occur regularly in nature; clouds and coastline are good examples of this. The coastline, as seen from space, appears as a rugged feature formed from inlets, bays, and peninsulas. If one takes a closer look, one sees a similar set of curves formed from sand and rocks. One of the most famous fractals in Computer Graphics is the Mandelbrot Curve.

The Mandelbrot Curve is formed by calculating a sequence of values f_0, f_1, f_2, \dots for each point (x, y) in a grid. The sequence f_0, f_1, f_2, \dots for each point (x, y) is formed by iteratively calculating the equations:

$$d_0 = 0 + 0i$$

$$c = x + yi$$

$$i = \sqrt{-1}$$

$$d_{k+1} = d_k^2 + c$$

$$f_k = |d_k|$$

For each point (x, y) , the sum d_{k+1} is iteratively calculated to determine whether d_{k+1} grows arbitrarily large. If d_{k+1} becomes greater than 2 at any point during the iteration, then successive iterations will then exponentially grow this value towards infinity and the iteration is stopped. This point is not considered part of the Mandelbrot curve, and is colored black. Otherwise, the iteration is repeated N times, where N is a suitably large value typically around 1,000. This point is considered part of the Mandelbrot curve and assigned a color based on its magnitude.

The Mandelbrot Curve algorithm was coded using the master/slave paradigm. In this paradigm, the first process created is designated as the master process, and all other processes are designated as slaves. The master process is responsible for assigning work to slave processes and accumulating the results. The master process sequentially assigns each slave a 400 square pixel area of the grid to process. The assignment arrives in the form of a four-element integer array that contains the (x, y) co-ordinate of the upper-left hand corner of the work region, and the (x, y) co-ordinate of the lower-right hand corner. The slave calculates the Mandelbrot value for each point over this region, iterating up to 1,000 times for each pixel. After the iteration over the work area completes, the slave sends a response back to the master in the form of a four-element integer array, followed by a two dimensional double array. This four-element integer array contains the co-ordinates of the work area as described above. The two dimensional double array contains the Mandelbrot curve values for each point within the work region. The master process copies the Mandelbrot values from this array into their respective positions in the 512 x 512 master pixel array. The master reassigns to the slave another section of the

grid to compute Mandelbrot values in. Processing completes when the Mandelbrot values for the entire 512 x 512 pixel array have been computed. Pixels are colored according to their Mandelbrot value. The output of this algorithm is a Sun raster file shown in Figure 7.

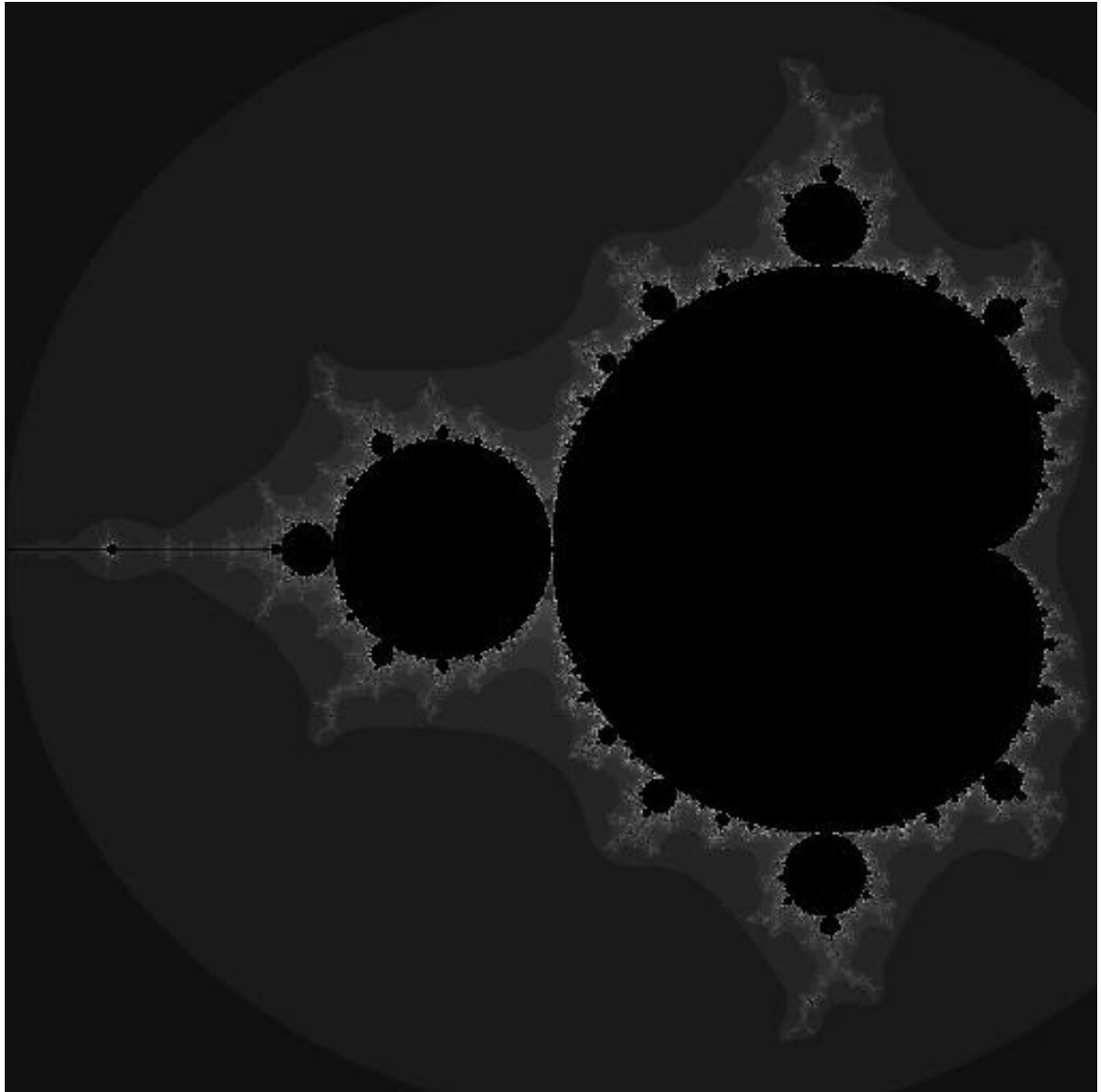


Figure 7. Output of the Mandelbrot curve application.

The performance of JMPI is determined by measuring the amount of time required to calculate the Mandelbrot values for the 512 x 512 pixel grid. For each test case, the elapsed time from three runs is averaged to produce a final result. The performance of JMPI is compared against MPICH and mpiJava for three test environments: distributing the master and slave processes evenly on the dual-processor and single-processor, distributing all processors within the dual-processor and distributing all processes within the single-processor. In addition, the number of slave processes is varied from one to five. Shorter elapsed times indicate better performance.

Figure 8 shows the amount of time elapsed calculating the Mandelbrot fractal with processes distributed over a fast Ethernet network for applications using MPICH, mpiJava and JMPI. The Java based implementations, mpiJava and JMPI, are significantly slower than the native C implementation with MPICH. Although the performance of JMPI is faster than mpiJava for one slave process, a linear slowdown occurs as each new slave process is added. Furthermore, the performance of mpiJava increases as the number of slaves increases from one to two, and the performance remains relatively constant as the number of slave processes increase from three to five. Although one would expect performance to decrease as new slave processes are added, the master process quickly becomes a bottleneck. Specifically, this bottleneck occurs as a result of copying the Mandelbrot results that are returned from the slave into the main pixel array. The linear slowdown in performance of JMPI is attributed to the overhead of Remote Method Invocation, Object Serialization and the overhead due to small message sizes.

Time (in seconds) to complete 512x512 pixel Mandelbrot Fractal over ethernet

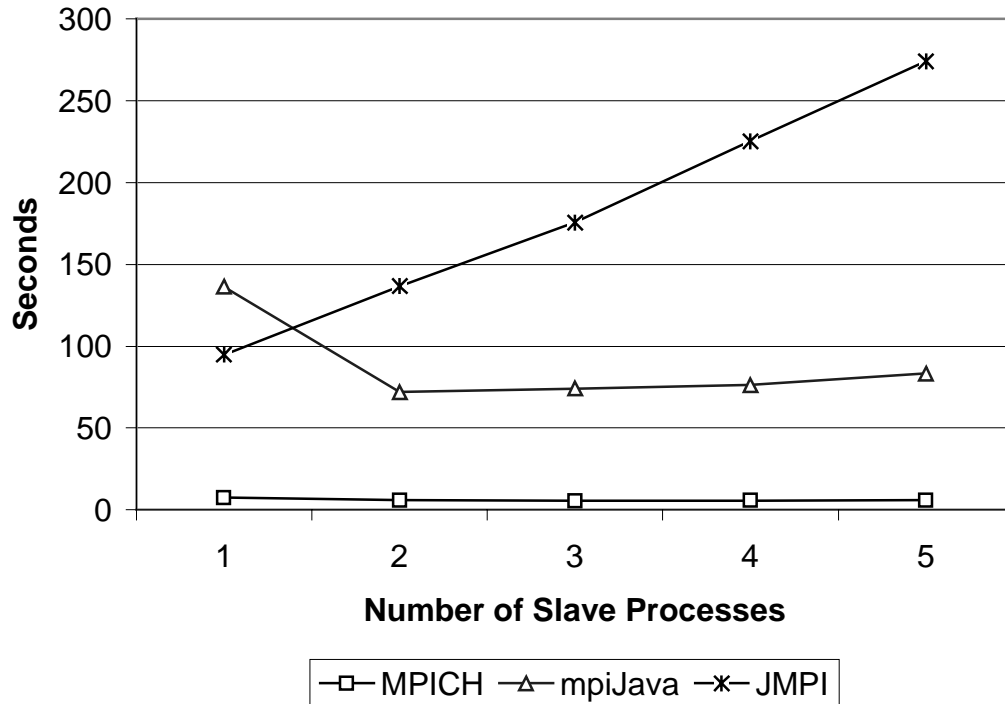


Figure 8. Performance of the Mandelbrot application over fast Ethernet.

Figure 9 shows the amount of time elapsed calculating the Mandelbrot fractal with processes distributed within a dual-processor Solaris X86 system. MPICH and mpiJava use the shared memory transport for passing messages and both perform significantly better than JMPI which uses RMI for passing messages. Although JMPI shows a performance increase when the number of slaves is increased from one to two, a linear slowdown occurs for each slave added after the second. Figure 10 shows the amount of time elapsed calculating the Mandelbrot fractal with processes distributed within a single processor Linux system. The degradation in performance is exhibited due to the decrease in processing power and the change in the message transport from shared memory to

sockets. Although JMPI performs significantly better than mpiJava when the number of slaves is less than three, JMPI performs worse than mpiJava as the number of slaves increases beyond four due to the linear slowdown in performance caused by the bottleneck at the master process. This bottleneck is attributed to the overhead of RMI.

Time (in seconds) to complete 512x512 Mandelbrot Fractal on dual-processor

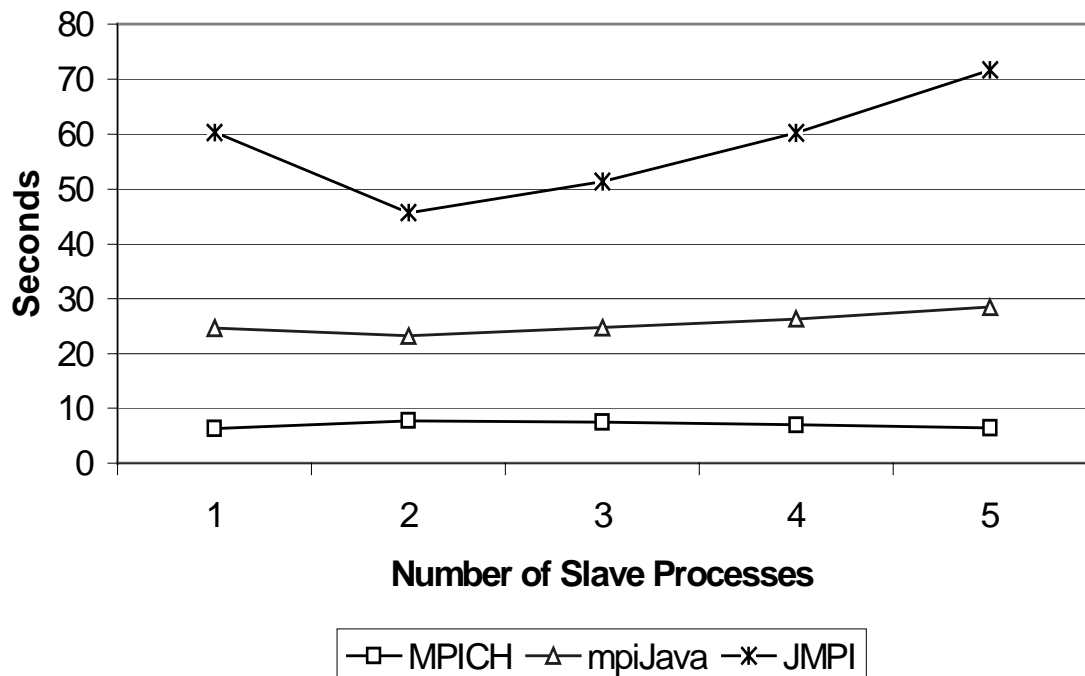


Figure 9. Performance of the Mandelbrot application within a dual-processor system.

6.4 Nozzle

Nozzle is a Java applet that simulates a 2-dimensional flow through an axisymmetric nozzle. At the core of the simulation is a four-stage Runge-Kutta time-stepping algorithm. The Runge-Kutta algorithm is used to solve Ordinary Differential Equations

whose initial variables are known and subsequent variables are guessed and iteratively calculated. The initial conditions of the flow are initialized from the “Tran0.dat” datafile distributed with the application. Additionally, a numerical dissipation algorithm is used to dampen spurious oscillations that could cause the solution to diverge in the presence of shock waves. The sequential version of Nozzle was developed by Saleh Elmohamed at the Northeast Parallel Architecture Centre (NPAC) and modified by Sang Lim at NPAC to use MPI. Nozzle is distributed in the examples subdirectory of mpiJava.

Time (in seconds) to complete 512x512 Mandelbrot Fractal on single processor

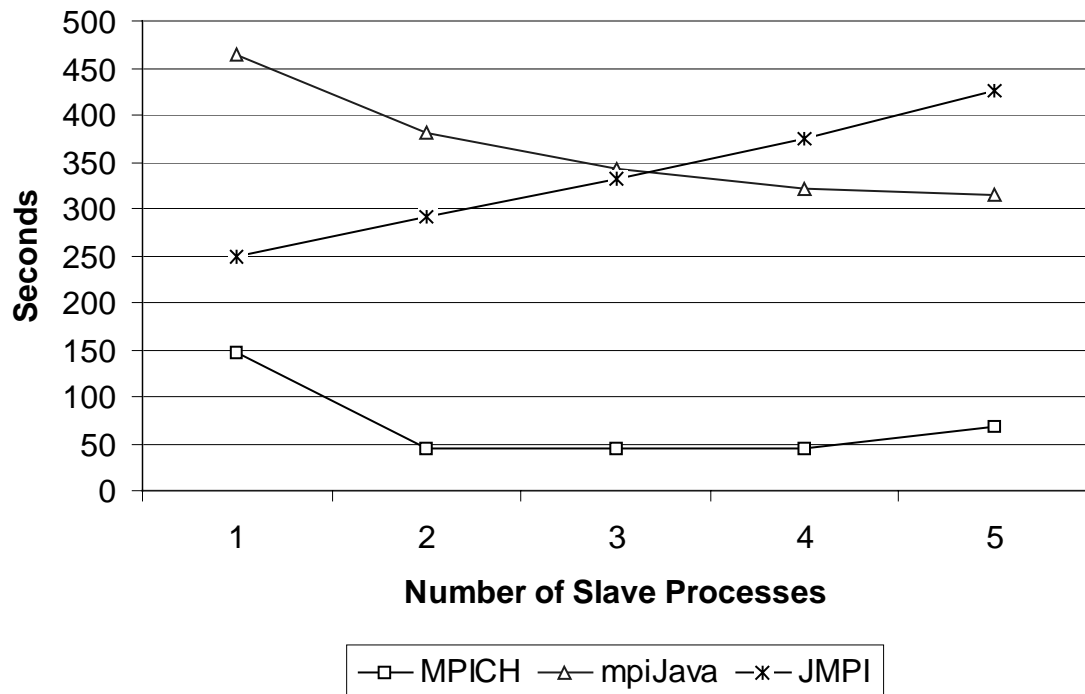


Figure 10. Performance of the Mandelbrot application within a single processor system.

Nozzle is a Java applet designed to run on n processes, where each process is responsible for $1/n$ of area within the nozzle. The rank 0 process has the additional responsibility of reading the starting conditions from the datafile and displaying results. The simulation displays the contour plots of six flow control variables in real-time: velocity, pressure, mach number, density, entropy, and temperature. Figure 11 shows the output of the Nozzle simulation for the flow-variable pressure contour-plots after several minutes of simulation. The flow through the nozzle is from left to right, and the boundaries between each process are shown with a solid line. Approximately 2,000 iterations are required if the input and output flow conditions are supersonic and approximately 6,000 iterations are required if the ingress and egress flow conditions are subsonic. The flow is considered subsonic when the speed of the fluid through the nozzle is slower than the speed of sound and supersonic when the speed of the fluid through the nozzle is greater than the speed of sound.

The purpose of the nozzle simulation is to show JMPI's adherence to the draft standard and show compatibility between the implementation of mpiJava and JMPI. The simulations were run each two processes in both mpiJava and JMPI in three test environments: two nodes connected via fast Ethernet with a single process on each, a dual-processor system with both processes, and a single-processor system with both processes. On the dual-processor system, each process ran concurrently on a separate processor. The native implementation used with mpiJava was MPICH. The source code required no modifications and was compiled with both implementations. The results of this simulation show problems with portability in the mpiJava environment. Specifically,

we were unable to get Nozzle to work correctly in all but the dual-processor environment. The MPICH backend reported message size mismatches between the sender and the receiver. However, we were able to get Nozzle to work correctly in the three test environments. Although no formal mechanisms exist for measuring the performance of Nozzle without modification of the source code, the two applications approached the same solution in the dual-processor environment in the same amount of time. Additionally, updates in the output on the screen appeared to happen within the same time interval in both platforms.

6.5 Conclusion

JMPI performance clearly suffers as a result of using Remote Method Invocation as the primary message transport between processes, specifically in the multi-processor environment. Although the semantics of Remote Method Invocation offer simpler implementations of some MPI primitives such as process barriers and synchronous communication modes, the performance of message transport would be greatly enhanced with the use of sockets. In addition, further research is needed to determine the performance benefits of using integrating multiple processes within the same Java Virtual Machine environment in a multi-processor system. An initial concern of such research would be the stability of multiple processes in the event that one process hogs resources. Although the performance of JMPI lagged behind mpiJava in most cases, the environment was incredibly robust and consistent as compared to mpiJava. If the changes made to the message transport of JMPI were made to JMPI, JMPI would perform significantly better than mpiJava.

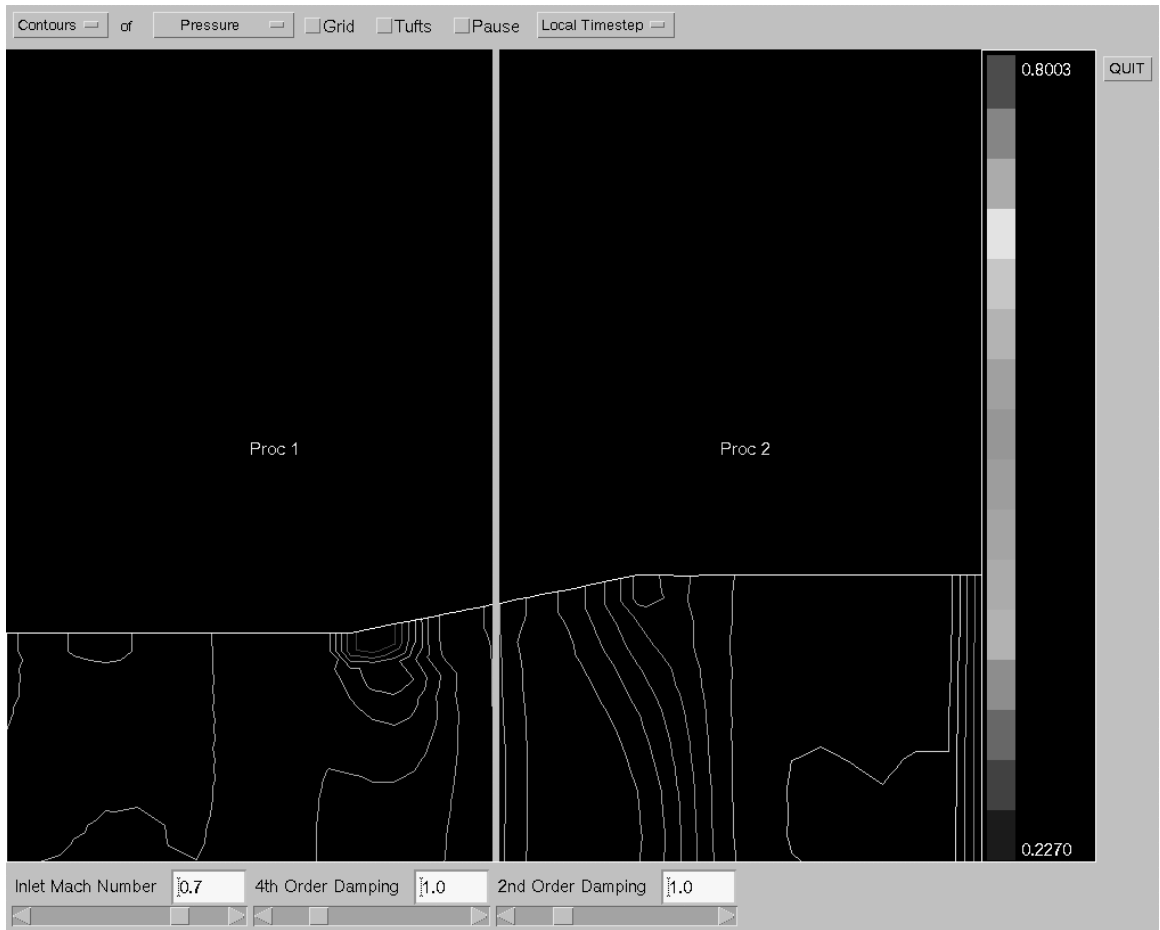


Figure 11. A screenshot of the Nozzle application.

CHAPTER 7

CONCLUSIONS AND FUTURE RESEARCH

7.1 Conclusions and Future Research

Although JMPI is a good starting point for a reference implementation of the Message Passing Interface for Java, the lack of performance needs to be addressed. Specifically, Remote Method Invocation (RMI) is too slow for handling message passing between processes. RMI adds a constant 325 bytes to the size of each message transferred and adds an average of 0.94 ms to message latency. A future research goal is to replace RMI in the Communications Layer with Berkeley Sockets. Although Berkeley Sockets would replace the bulk of message transfer, RMI would still play a role in virtual machine initialization and process synchronization. A second research goal is to run multiple processes as separate threads within one Java Virtual Machine on the same physical machine. Currently, one JVM is instantiated for each process started in the virtual machine. When multiple processes run within the same Java Virtual Machine, shared memory would be used to exchange messages between processes. Shared memory would increase the performance of message transfer between processes while decreasing the total memory required per machine.

APPENDIX

GETTING STARTED WITH JMPI

A.1 Introduction

This appendix describes the process of downloading and installing the JMPI software, configuring a virtual machine, and compiling and running a sample application supplied with JMPI. JMPI is completely written in Java, and requires a Java Development Kit (JDK), version 1.2 or better to run. In addition, users of JMPI on the Microsoft Windows platform will also need to install and configure a remote shell (RSH) daemon on their machines.

A.2 Downloading the JMPI software

The latest version of the JMPI source code, pre-compiled class files, and documentation is available on the web at:

<http://euler.ecs.umass.edu/jmpi>

JMPI is distributed as a compressed tar archive for UNIX platforms, and as a WinZip file for Microsoft Windows platforms. Unpacking the archive creates `mpi.jar` in the top-level directory, and `src` and `doc` sub-directories. The `mpi.jar` is a pre-compiled archive of the classes that form the `mpi` package, and is ready for immediate use. A Makefile is included in the `src` directory, running `make` will recompile the source code and recreate the `mpi.jar` archive. The batchfile, `makefile.bat` is included for users recompiling on the Microsoft Windows platform.

A.3 Downloading the Java Development Toolkit

JMPI requires version 1.2 or better of the Java Development Toolkit (JDK) or Java Runtime Environment (JRE). Version 1.3, which is currently in active beta at the time this document is being written, is preferred over version 1.2 because of the performance enhancements made to Remote Method Invocation and Object Serialization. This version of JMPI, 1.0, is known to work with the JDK platforms listed below. The JDK is available free of charge.

For users of Microsoft Windows 95/98, Windows NT, and Windows 2000, Sun Solaris for SPARC, and Sun Solaris for Intel, the recommended JDK is available for download from Sun Microsystems website:

<http://www.javasoft.com/>

At the present time, Sun has released final versions of the Java 2 Software Development Environment Standard Edition (J2SE) for the platforms listed above. In addition, a public beta of version 1.3 of J2SE is available to users on the Microsoft Windows platform. If you plan to run MPI on the Windows platform and need an RSH daemon, one is available at:

<http://www.denicom.com/>

For users of Linux on the x86 platform, there are two versions of the Java Development Toolkit known to work with JMPI. The first is Blackdown JDK version 1.2.2rc4, ported directly from Sun's source code, and is available at:

<http://www.blackdown.org/>

The second is a beta release of the JDK version 1.3 from IBM's Alphaworks project and is available at:

<http://www.alphaworks.ibm.com/>

For a complete list of JDKs available for other platforms, visit:

<http://java.sun.com/>

A.4 Configuring the MPI runtime environment

The set of hosts that a MPI application will execute on forms a virtual machine. Before an MPI process is compiled or executed on the virtual machine, two environmental variables need to be set or modified. In addition, each host in the virtual machine needs to have an entry in the .rhosts (or .shosts) file of the host that launches the MPI application with mpirun. If your home directory is not mounted on all hosts within the virtual machine, make sure to install JMPI and set the environment variables on each host appropriately.

The JDK uses the CLASSPATH environment variable to locate Java classes. The filename of the jmpi.jar archive and the path to the MPI application need to be included in the CLASSPATH environment variable. For example, if the jmpi.jar is located in /usr/local/lib, and you wish to execute MPI class files in your home directory, set the CLASSPATH environment variable as follows:

```
% setenv CLASSPATH /usr/local/lib/jmpi.jar:
```

The PATH statement needs to include the PATH where the JDK binaries are installed. For example, if the JDK is installed in /usr/local/jdk1.22, include /usr/local/jdk1.22/bin in

your PATH statement. If your home directory is mounted on all hosts within the virtual machine, set the environment variables in your .cshrc or .profile file.

A.5 Creating the machinefile

The file named *machinefile* contains an entry for each host which will participate in the virtual machine. Each host is placed on a single line, optionally followed by the number of processes to run on that host. If omitted, one process is started per machine. Additionally, the user may optionally specify the path and filename of the JAVA class on the remote machine. Finally, the user can optionally disable the Just-In-Time (JIT) compiler. Although disabling JIT compilation results in significantly slower performance, stack traces will include the line number in the source code that caused the exception. Lines beginning with a # are considered comments and ignored.

```
golden-gate 1 /usr/home/fred/master
embarcadero 2 /export/home/fsmith/slave
```

Figure 12. Example virtual machine configuration file

Figure 12 shows an example virtual machine configuration file. Three processes are started in total, one process of /usr/home/fred/master class on the machine golden-gate, and two processes of /export/home/fsmith/slave on the machine embarcadero.

A.6 SendMessages.java : a sample Java MPI Application

Figure 13 shows the source code for a trivial three-process MPI application. Process with rank 0 forwards the contents of float array `buf1` to the rank 1 process, and also forwards the same float array to the rank 2 process. The rank 1 process waits for the float array to arrive from the rank 0 process, negates each element, and forwards the array to the rank 2 process. The rank 2 process waits for a float array from the rank 0 process, and a float array from the rank 1 process. Rank 2 prints the contents of both arrays, and all processes in the application terminates.

A.7 Compiling SendMessages

If you have modified your `PATH` and `CLASSPATH` environment variables as described in section A.4, then you are ready to compile `SendMessages.java`. To compile `SendMessages`, execute the following command:

```
% javac SendMessages.java
```

If there are no typos in the `SendMessages.java` source file, the compiler outputs `SendMessages.class`.

A.8 Running SendMessages

JMPI applications are started with `mpirun`. If the virtual machine configuration file has been created as described in section A.4, and the host running `mpirun` has been added to the `.rhosts` (or `.shosts`) file on all hosts in the virtual machine, then the following command starts the MPI application:

```
% java mpi.mpirun -np 2 SendMessages
```

mpirun reads machinefile to determine the hosts which will participate in the virtual machine, and launches individual processes on remote hosts using either rsh or ssh. While the remote processes are running, stdout and stderr are redirected to the console mpirun is executing on. The virtual machine can be aborted with hitting Control-C. mpirun concludes when all processes in the virtual machine exit.

```

import java.io.*;
import mpi.*;

public class SendMessages {

    public static void main(String [] args) throws MPIException {

        int size;
        int rank;
        int offset = 0;
        int count = 2;
        double [] buf1 = new double[2];
        double [] buf2 = new double[2];

        MPI.Init(args);

        size = MPI.COMM_WORLD.Size();
        rank = MPI.COMM_WORLD.Rank();

        if (rank==0) {
            buf1[0] = (double) 3.141;
            buf1[1] = (double) 2.718;
            MPI.COMM_WORLD.Send(buf1, offset, count, MPI.DOUBLE, 1, 1);
            MPI.COMM_WORLD.Send(buf1, offset, count, MPI.DOUBLE, 2, 1);
        }
        if (rank==1) {
            MPI.COMM_WORLD.Recv(buf2, offset, count, MPI.DOUBLE, 0, 1);
            buf2[0] = -buf2[0];
            buf2[1] = -buf2[1];
            MPI.COMM_WORLD.Send(buf2, offset, count, MPI.DOUBLE, 2, 1);
        }
        if (rank==2) {
            MPI.COMM_WORLD.Recv(buf1, offset, count, MPI.DOUBLE, 0, 1);
            MPI.COMM_WORLD.Recv(buf2, offset, count, MPI.DOUBLE, 1, 1);
            System.out.println(buf1[0] + ":" + buf2[0]);
            System.out.println(buf1[1] + ":" + buf2[1]);
        }
        MPI.Finalize();
    }
}

```

Figure 13. SendMessages.java Example Java MPI Application

BIBLIOGRAPHY

- Baker, M., Carpentar, B., Fox, G., Ko, S. H., and Lim, S. mpiJava: “An Object-Oriented Java interface to MPI,” November, 1999.
- Carpentar, B., Fox, G., Ko, S. H., and Lim, S. “mpiJava 1.2 API Specification,” November, 1999.
- Carpentar, B., Fox, G., Ko, S. H., Li, X., and Zhang, G. “A Draft Binding for MPI,” September, 1998.
- Carpentar, B., Getov, V., Judd, G., Skjellum, A., and Fox, G.. “MPI for Java: Position Document and Draft API Specification,” *Java Grande Forum*, 1998.
- Chan, P. The Java Developers Almanac. Reading, Massachusetts: Addison Wesley Longman, 1998.
- Flanagan, D. Java in a Nutshell, second edition. Cambridge: O’Reilly & Associates, 1997.
- Flanagan, D. Java Examples in a Nutshell. Cambridge: O’Reilly & Associates, 1997.
- Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. PVM: Parallel Virtual Machine: A Users’ Guide and Tutorial for Networked Parallel Computing. Cambridge, Massachusetts: The MIT Press, 1994.
- Getov, V., Flynn-Hummel, S., and Mintchev, S. “High-Performance Parallel Programming in Java: Exploiting Native Libraries,” Harrow, England and Yorktown Heights, NY: October, 1997.
- Gordon, R. Essential JNI Java Native Interface. Upper Saddle River, NJ: Prentice Hall PTR, 1998.
- Gosling, J., Joy, W., and Steele, G. The Java Language Specification, Version 1.0. Reading, Massachusetts: Addison Wesley Longman, 1996.
- Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W. and Snir, M. MPI-The Complete Reference: Volume 2, The MPI-2 Extensions. Cambridge, Massachusetts: The MIT Press, 1998.
- Gropp, W., Lusk, E., and Skjellum, A. Using MPI: Portable Parallel Programming with the Message Passing Interface. Cambridge, Massachusetts: The MIT Press, 1994.

Java Grande Forum Panel. Java Grande Forum Panel Report: Making Java Work for High-End Computing. Orlando, Florida: November, 1998.

Message Passing Interface Forum. A Message Passing Interface Standard. March, 1995.

Message Passing Interface Forum. Extensions to the Message Passing Interface. August, 1997.

Mintchev, S. "Writing Programs in JavaMPI," London, England: University of Westminster, October, 1997.

Oaks, S., and Wong, H. Java Threads. Cambridge: O'Reilly & Associates, 1997.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. MPI-The Complete Reference: Volume 1, The MPI Core, second edition. Cambridge, Massachusetts: The MIT Press, 1998.

Zhang, G., Carpenter, B., Fox, G., Li, X., and Wen, Y. "Considerations in HPJava language design and implementation," Syracuse, New York: Northeast Parallel Architectures Centre, Syracuse University, October, 1998.

