JAVA BINDINGS TO PARALLEL VIRTUAL MACHINE

by

Eric Fialkowski

A project

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

Spring 2004

The project presented by *Eric Fialkowski* entitled *Java Bindings to Parallel Virtual Machine* is hereby approved.

Amit Jain, Advisor	Date
John Griffin, Committee Member	Date
Jyh-haw Yeh, Committee Member	Date
Jack Pelton, Graduate Dean	Date

This is dedicated to my kids (Chelsi, Logan, Cody, and Lacey) and the rest of my family. Without their support, understanding, and sometimes pressuring I would not have been able to accomplish this.

ACKNOWLEDGEMENTS

Thank you to all of my professors for preparing me to undertake this. Without the knowledge you helped me garner throughout my college career, I would never have been able to complete an undertaking such as this. I would also like to thank my family again for helping me tackle and complete this endeavor.

ABSTRACT

Java bindings to Parallel Virtual Machine allows those who know Java to start programming for Parallel Virtual Machine without learning a new language. This allows them to focus on the parallel aspects. The binding will also allow new applications for parallel programming by exposing all of the libraries that Java provides. While there a multitude of other languages and parallel communication libraries, this combines a well known language with a well known library.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Introduction

1.1.1 Background on clusters

Computing clusters are not a new concept. Large minicomputer/mainframe class computers had clustering capabilities. A new class of cluster computer using cheaper components, so called COTS - commercial of the shelf components, has emerged allowing greater access to high-end computing power. The *Beowulf* cluster pioneered by Donald Becker and Thomas Sterling at NASA's Goddard Space Flight Center [1] combines commonly found components such as the PC and Ethernet for a cost-effective high-performance computing resource.

1.1.2 Background about Java

Java was developed by Sun Microsystems to be a portable computing language running on everything from embedded devices to high-end computers [2]. The language has many features that help in developing programs including memory management and *forced* error handling. This with an extensive built-in library allows a developer to write many different types of programs without having to learn vastly different programming paradigms. Java's portable nature also allows a programmer to develop on

a lower cost workstation and have the end result run a a much higher-end computer.

1.1.3 Background about PVM

Parallel Virtual Machine (PVM) is a library that allows multiple, smaller computers to be networked together to act as a single, parallel computer. To quote the PVM website: [3]

> PVM (Parallel Virtual Machine) is a software package that permits a heterogeneous collection of Unix and/or Windows computers hooked together by a network to be used as a single large parallel computer. Thus large computational problems can be solved more cost effectively by using the aggregate power and memory of many computers. The software is very portable. The source, which is available free through netlib, has been compiled on everything from laptops to CRAYs.

1.1.4 Using Java, PVM, and Beowulf Clusters

It only seems fitting that Java and PVM combine. The cross-platform nature of Java allows clusters of heterogeneous machines to be constructed, taking advantages of the given platform's strengths. The error handling and memory management aspects to Java also helps in writing well-behaved code. PVM also allows heterogeneous programs to be written when it is necessary to get the highest performance possible or to tie to legacy applications.

1.1.5 Prior works

There have been two other attempts to combine PVM and Java. One which serves as a major basis for this project and another which provides PVM functionality but is a Java-only solution.

jPVM

The project jPVM [4] serves as the foundation for the work of PVM Bindings in Java (henceforth refered to as PBJ.) It is a simple JNI wrapper that provides a single object for accessing PVM functionality. The work in this project is leveraged into multiple objects and a more object-oriented design.

JPVM

The JPVM project [5] (not to be confused with the lowercased "jPVM") is a PVM-like, all Java environment. From the project website:

> JPVM is a PVM-like library of object classes implemented in and for use with the Java Programming language. PVM is a popular message passing interface used in numerous heterogeneous hardware environments ranging from distributed memory parallel machines to networks of workstations. Java is the popular object oriented programming language from Sun Microsystems that has become a hot-spot of development on the Web. JPVM, thus, is the combination of both - ease of programming inherited from Java, high performance through parallelism inherited from PVM.

While JPVM is an impressive undertaking, native PVM compatibility was desired so it was not well suited for the problem at hand. Connecting to standard C programs written for PVM is a desired trait that JPVM does not facilitate. Programs written with JPVM are only compatible with other programs written in JPVM.

1.1.6 Goals

This project has some modest goals. Create a library for writing programs in Java that use the PVM system for communication. The library should be cross platform and should be able to interact with PVM programs written in other languages (C being the litmus test.) The library should follow standard Java programming paradigms, where they make sense.

Chapter 2

ADVANTAGES OF JAVA AND PVM

2.1 Java's Advantages

Java was designed as an object-oriented program language with various features for building robust applications. Some features, such as Exceptions, help in writing correct applications while other features help in connecting to databases or writing graphical applications. Java was also designed to work on a variety of platforms which can work in conjunction with PVM's heterogeneous nature.

2.1.1 Memory Management

Java handles memory allocation and management removing an error prone task from the programmer. Memory errors can still occur but are not as esoteric as memory management errors in C.

2.1.2 Exceptions

Java's exception routines help by taking a heavy-handed approach to force developers to handle errors. It is still possible to ignore the exception by having an empty code block in the catch section but the program will not compile without the try-catch mechanism.

2.1.3 Built-in support for threads/db connections/networking/graphics

Java has built-in support for threads which are useful for programs running on multi-processor computers. Network communication can be regulated by writing a multi-threaded program instead of spawning multiple processes and this can improve parallel performance in some cases. JDBC allows for connecting to relational databases and provides an abstraction for connecting to various databases.

Why is built-in better than external libraries?

There are many add on packages for C that provide graphics and threading but by Java's inclusion of them in the standard libraries, there are not new programming paradigms to learn. In the case that the user's main role is not programming (perhaps a scientist who needs to program a simulation) the inclusion leads to one less thing to learn.

2.1.4 Java's Use in Computer Science Curricula

In a majority of computer science programs, much of the curriculum is being taught with Java. By integrating Java and PVM, courses on parallel and distributed computing do not have to teach a whole new programming language, such as C. That leaves more time in the course to focus on the features of PVM and parallel computing.

2.2 Why PVM

PVM has the advantage that it has been around and is a tried and true set of libraries. There are many ways to write a parallel program but PVM has abstracted the low level details out. A programmer can focus on the algorithms and not on the details such as opening and closing sockets or group membership. PVM provides dynamic task creation enabling the number of processors working on a given problem to grow or shrink as needed. Tasks are uniquely identified so that messages can be passed by task ID without knowing anything about where the task is running. PVM is also designed to work on with heterogeneous networks, which matches Java's cross platform nature.

2.2.1 Pros and Cons for alternatives to PVM for parallel programs

PVM is not the only way to write parallel programs. There are many different ways, each with their own advantages and disadvantages. See Table 2.1.

TABLE 2.1 Alternatives for writing parallel programs in Java

Method	Pros	Cons
sockets	cross platform and cross programming language. Very little overhead.	very verbose programs. Very few details are hidden from the programmer.
JMS	defined standard. Can pass objects around.	Single language. Much overhead.
CORBA	cross platform and cross programming language	Complex, implementations do not inter-operate very well
RMI	work at the object level.	Single language, high network traffic overhead
MPI	Well defined standard.	No cross-implementation interoperability. Designed by community.
SOAP	cross platform and cross programming language. XML based communication.	Bandwidth inefficient. Interoperability issues.
Grid Computing [15]	SOAP-based parallel and distributed computing targetted for ad-hoc clusters	Very new technology.

Chapter 3

DESIGN OF THE SYSTEM

3.1 Core Design Decisions

PBJ is designed to be an object-oriented abstraction of PVM. It should follow standard Java paradigms to ease developer transition from single-system programming to parallel programming. At no point should the recognized standards for either Java or for PVM override the need to break from said changes. The combination of PVM and Java presents new possibilities and therefore will require new standards to be set. There are a few ideals that should be followed, which are outlined below.

1. Must be thread-safe.
2. Exceptions will be used to propagate errors.
3. As close as compatibility as possible with C-based PVM.
4. Separation of functionality into classes.

3.2 Identifying components

The first step is to identify what objects exist in the system. Figure 3.1 shows the basic components for PVM. PVM is based on the concept of sending messages to tasks.

`PVMTask` fulfills the task role and PVMMessageBuffer encapsulates the messages. PBJ departs slightly from standard PVM in that there is no concept of a default message buffer. In reality, the default message buffers in PVM are not different from any other message buffer. There are some behind-the-scenes conveniences occurring to make it look like only one message buffer is active. [1] Other major components identified include `PVMException`, `PVMGroup`, and `PVM`.

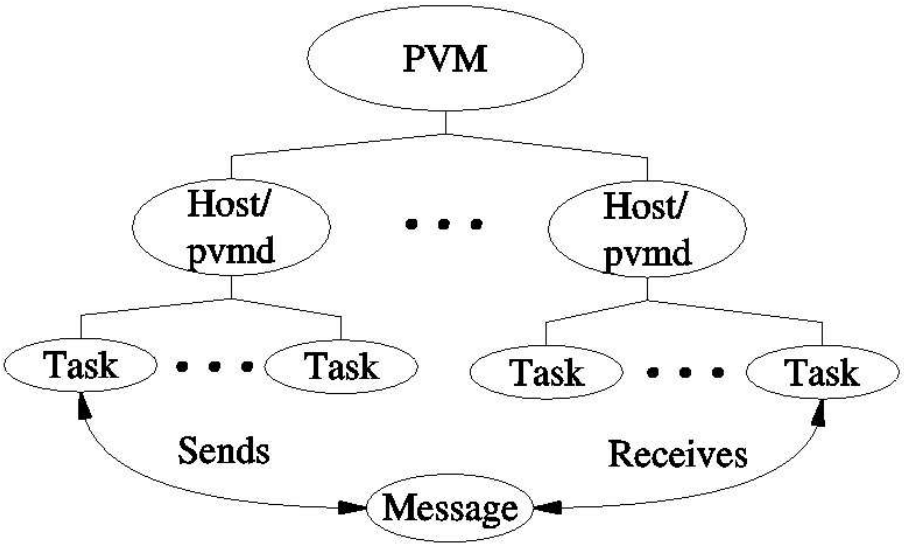

Figure 3.1. High Level Conceptual Overview of the PVM system

3.2.1 Design of PVMTask

`PVMTask` represents the main execution unit in a parallel program. Any task running in PVM must extend this task. This object mainly contains methods for receiving messages. A task in PVM does some basic things: enrolls into PVM, exits from

[1] `pvm_initsend()` creates a new buffer and deletes the old versus actually clearing out the old buffer.

PVM, sends messages, receive messages, and processes data in the messages. The `PVMMessageBuffer` class, which handles sending and processing messages, is unique for each individual program (and hence the need to extend this class for tasks.) The `PVMTask` object then is responsible for enrolling into PVM, exiting from PVM, and receiving messages. There are some notification routines that also fall under receiving messages. In order to help Java programmers migrate to writing PVM programs, the concept of a listener for messages was created. Many Java programs are event-driven, waiting for some input to act upon, so PBJ should mimic this behavior. In order to achieve that, Java threading is utilized. PVM does not have a way to register callbacks for when a given message is received. It would be very difficult to extend callbacks from C into Java even if it did. PVM does however have a method to look for a message waiting to be received. The PVM call "pvm_probe()" is used from within a thread to see if there are any messages to be received. Event-driven message reception is best suited for when the task needs to receive a various amount of messages that can have a variety of different message tags. Such tasks are typically the controller/master task for a given program.

TABLE 3.1 Major Methods for PVMTask

Method Name	Purpose
myTID	Returns the task's TID in the system
exit	Exits task from PVM system
receive	Receives a PVMMessageBuffer, blocking
nonBlockingReceive	Receives a PVMMessageBuffer, non-blocking
timedReceive	Receives a PVMMessageBuffer, blocks for timeout
enableMessageEvents	Starts polling thread to look for messages for PVMMessageBufferListener
disableMessageEvents	Stops the polling thread

Design of PVMMessageBufferListener

This interface is designed to allow `PVMTask` to decide which messages to listen for in the asynchronous model and to provide a callback for processing. It is an interface with the callback method and two selector methods. The selector methods are used to state which TID and which message tags to accept and process messages for. A class that wishes to process messages asynchronously needs to provide an implementation of this interface.

TABLE 3.2 Major Methods for PVMMessageBufferListener

Method Name	Purpose
lookingForMessageTag	Returns the message tag to listen for
lookingForTID	Returns the TID to listen for
receivedPVMMessageBuffer	Called when a message matching the tag and TID is received (must override)

Design of PVMMessageBufferAdapter

This object is an abstract object that provides basic implementations of the selector methods of `PVMMessageBufferListener`. These default to any message from any task. This deviates slightly from Java adapter classes in that the callback method still needs to be overridden. This is due to the fact that the only reasonable default processing would be to ignore the incoming message. This can just as easily be accomplished by not receiving any messages (although there may be a build up of waiting messages in the PVMD.)

TABLE 3.3 Major Methods for PVMMessageBufferAdapter

Method Name	Purpose
lookingForMessageTag	Returns -1, the wildcard
lookingForTID	Returns -1, the wildcard
receivedPVMMessageBuffer	Called when a message matching the tag and TID is received (must override)

3.2.2 Design of PVMMessageBuffer

This class is really the work-horse of the system. Messages are the key component in a message-passing system like PVM so most of the work happens here. Java's method overloading allows for some ease of use from the standard PVM message packing routines. In standard PVM one must pack arrays and include the size of the array (a C limitation Java overcomes) and the *stride* of the array. The stride is which elements to include from the array. A stride of one means every item, two means every other item and so on. So even when sending one item, the call includes the size and stride (both set to one.) Java allows for methods that will handle setting the trivial values for the programmer. Method overloading also allows for a single function name for packing any of the different types. The data type is required to be known to unpack data from a message. Java does not allow for the same method name to return different data types. A `PVMMessageBuffer` also contains the methods for sending itself.

Working with datatypes

Java and C have a slightly different primitive datatype set, with Java's being more restrictive at the same time as being more exact. Java's datatypes are all mapped to the underlying C routine so messages can contain any of int, short, long, float, double,

char, byte, and boolean. The boolean is converted from true/false values to one and zero. This allows boolean values to be sent between C and Java. There is also a need to send Java objects for pure Java applications. Java's built in remote method invocation(RMI) uses a special type of object that can be sent across the network. The `java.io.Serializable` interface indicates that an object can be turned into a byte array that can be transfered across the network. This concept is well known and works so it was extended into the PBJ system. RMI has the ability to send the class definition across in addition to the data. PBJ does not need this capability because the primary target operating environment is a Beowulf cluster where all programs will have access to the same classes.

TABLE 3.4 Major Methods for PVMMessageBuffer

Method Name	Purpose
pack	Places data into the buffer
unPack*	Retrieves data from the buffer
freeBuffer	Releases the buffer and all resources held by it
send	Sends the buffer to the given task
makeBuffer	Factory method to create a PVMMessageBuffer object

3.2.3 Design of PVMException

`PVMException` is the class that is used to signal any error returns from the PVM system. There are a variety of different error returns that can occur in the system. One design idea would be to subclass `PVMException` for each different exception type. This would follow a more strict object-oriented design idiom but could create extra work for the programmer. For example, `pvm_send()` has three error returns, one which is not that applicable to PBJ. If multiple exception types where used, there

```
try {
}catch (IOException ioex) {
}catch (PVMPvmBadParamException pvmbpex) {
}catch (PVMPvmSysErrException pvmsyserrex) {
}
```

Figure 3.2. Verbose Catch Statements

would either need to be a catch block for each unique condition to handle or a catch for the base class, then use `instanceof` to determine which condition occurred. It was decided that having the cause in the exception and use only one exception type would be just as useful. With a constant value in the exception, a switch case can be used in the catch block when handling of the different exceptions is required. This has a performance benefit over the `instanceof`. Many different exceptions can occur in Java programs and by having just one type of exception for PBJ, code clutter can be reduced. For example, in Figure 3.2 shows a verbose catch for each type of Exception. Figure 3.3 shows a more typical condensed set of catch statements. When handling is different for a certain error condition, the code in Figure 3.4 can be used with higher efficiency than `instanceof`.

3.2.4 Design of PVMGroup

`PVMGroup` contains methods for group operations. These include the scatter, gather, broadcast and reduce operations. There are some advantages to wrapping an object

```
try {

}catch (IOException ioex) {

}catch (PVMException pvmex) {

}
```

Figure 3.3. Streamlined Catch Statements

```
try {

}catch (IOException ioex) {

}catch (PVMException pvmex) {
 switch ( pvmex.getReason() ) {

  case PvmBadParam:

   break;

  default:

   break;
 }
}
```

Figure 3.4. Streamlined Catch Statements with Breakout

around the group call. After a group is frozen, a flag is set in the object so no further calls into the PVM system are required. The communication methods are similar in nature to `PVMMessageBuffer` where there are overloaded methods for the various datatypes. One missing feature is the ability to pass custom functions to the reduce method. Callbacks from C to Java are difficult to write and add instability to the Java program. The broadcast differs from standard PVM by requiring a `PVMMessageBuffer` to be passed in instead of using the default send buffer.

TABLE 3.5 Major Methods for PVMGroup

Method Name	Purpose
getInstance	Returns which count in the group the caller is
gather*	Performs a gather operation on the group returning the specified type
reduce*	Performs a given reduce operation on the group returning the specified type
scatter*	Performs a scatter operation on the group returning the specified type
size	returns the current size of the group

3.2.5 Design of PVM

The `PVM` object represents the virtual parallel environment. There can only be one such environment so the object is implemented as a singleton, ensuring that one and only one instance of the object is ever created. The `PVM` class is responsible for functionality such as spawning new tasks, killing existing tasks, sending signals, and returning information about the current virtual machine. Most of the methods are just wrapper methods to native calls but Java allows for some convenience by way of function overloading. For instance, the *spawn* method has multiple versions based on how many extra parameters are needed. The spawn method also shows a point where

the method is more C-like than Java like. The method requires at a minimum the task name and an array to hold the spawned tasks' TIDs and returns the number of tasks actual spawned. It is not common to use parameters in Java to return data. A special spawn return type could have been created that contained the TID array and the number of successful spawns. It was decided that returning data in parameters was not that far of a break from standard Java coding practices and results in a little less overhead. To check if all expected tasks where spawned, it is a simple matter to check that the return equals the length of the passed in TID array. This should be the norm that they are equal. The library was optimized for the common case.

TABLE 3.6 Major Methods for PVM

Method Name	Purpose
spawn	start a new task
config	returns an array of `PVMHostInfo` for all hosts in the virtual machine
alltasks	returns an array of `PVMTaskInfo` for all tasks running in the virtual machine
kill	kills the given task
halt	shutdown the virtual machine and all tasks

3.2.6 Helper classes

There are some structures used in PVM that are ideal formats for the data they contain. The structures are mapped into Java objects. These objects are not very complex, which reflects the nature of the structures they represent.

PVMHostInfo

This is a representation of the `pvmhostinfo struct`. It is a simple class to hold the information that would normally be contained in the struct. The JNI call actually calls back into Java to create these objects.

PVMTaskInfo

This represents the `pvmtaskinfo struct`. It is returned from calls to the PVM environment requesting task information. The JNI call actually calls back into Java to create these objects.

TimeVal

This object mimics C's `timeval struct`. The `struct` is straightforward and there did not seem to be any benefit from doing anything more than simply porting it to a Java object. Various time or clock routines use this structure.

PVMObject

Java JNI libraries need to be loaded at run time. The library also should be loaded only once. To facilitate the single point of loading, a base class was created. This class has a static initializer to load the library. Any PBJ class that needs to access native libraries extends this class.

3.3 Tools used in construction

Various tools were used in the construction of PBJ. The tools reduced the complexity of building, testing, and debugging the project. Sun's Java Development Kit [12] was used to develop the code. Any of the Java 2 versions (1.2, 1.3, or 1.4) should work but development was done using the latest code. The code was developed primarily using Redhat [7] Linux but has also been tested on Microsoft [8] Windows. Windows would be an ideal target for development except configuring it to work with PVM is challenging. The lack of a built in RSH daemon also provides a barrier to the platform. The two differing platforms verified that PBJ is, indeed, cross platform.

3.3.1 Ant

Ant [9] is a build framework for Java. It is akin to `make` but is targeted for Java. It is a cross platform build environment, written in Java and customizable. It is based on the concept of tasks. Each task can depend on another task and there are a variety of tasks that can be used (such as compiling, copying files, and creating jar files.) Since PBJ uses C code in the JNI layer, a call to the command-line compiler was used. In newer versions of Ant there is a task for compiling C code.

3.3.2 JUnit

Junit [10] is a unit test framework. Used in conjunction with Ant, it can perform simple regression testing after every build. Tests for the basic functionality were created to validate that results were consistent. Many complex aspects for PBJ could be put into Junit test cases but was determined to not be as beneficial as just running

the basic tests.

3.3.3 Log4J

Log4J [11] is a logging library that abstracts out the what to log from the where to log it to. Log4J has the concept of message hierarchy, so various levels of information can be returned. Logging can go to various destinations - console, files, database, and even remote receivers. The hierarchy helps by separating development messages from error messages and can be configured without recompiling the program. The ability to log to a remote destination can be particularly useful with parallel and distributed programs.

Chapter 4

SAMPLE PBJ PROGRAM

This chapter will show a basic PBJ program and then a larger program that spawns a C program for computation.

4.1 Simple Example

There are a few basic sections in a PVM program.

1. Enroll in PVM
2. Send/receive messages
3. Process the messages
4. Exit PVM

Each of these steps will be explained where they are used in the timing code in Appendix A.

The first step is that any task needs to extend `pbj.PVMTask`.

Next, to enroll in PVM it is a simple call to `myTID()` (Figure 4.1.)

```
try
{
    System.err.println("My tid: " + task.myTID());
}
catch (PVMException pvme)
{
    pvme.printStackTrace();
    System.exit(1);
}
```

Figure 4.1. Enrolling in PVM

Creating a message and sending is a call to `PVMMessageBuffer.makeBuffer()`, followed by calls to `pack()` on the resultant buffer. The last step is to `send()` the buffer (Figure 4.2).

Receiving and unpacking a message is accomplished by doing a `receive()` operation from `PVMTask` to first receive a buffer and then calling the appropriate `unPack()` methods (Figure 4.3.) to copy the data into variables.

Exiting from PVM is a simple call to `exit()` from `PVMTask` (Figure 4.4).

4.2 Mandelbrot Example

The next example is a more complete example, taking advantage of Java's graphics and using a C program to do the calculations. The Mandelbrot set is a computation-

```
public static final int SIZE = 1000;

...

PVMMessageBuffer sender = null;

...

sender = PVMMessageBuffer.makeBuffer(PvmDataRaw);
data = new int[SIZE];

...

// build up what we're sending
for ( int i = 0 ; i < SIZE ; i++)
{
    data[i] = i;
}

...

sender.pack(data);

// send it off
sender.send(toTID, 1);
```

Figure 4.2. Sending a message

```
public static final int SIZE = 1000;

...

int data[] = null;

...

PVMMessageBuffer buffer = null;

...

buffer = task.receive(-1, 1);
data = buffer.unPackInt(SIZE);
```

Figure 4.3. Receiving a message

ally intensive program that falls under the "Embarrassingly Parallel" category [13], so it is easy to adapt to parallel operation. The sample program has three different ways that it can calculate the values. One is a sequential model, another uses multiple worker threads (for testing multi-processor machines), and the third spawns off PVM tasks to calculate the values.

The code for the PVM portion can be found in Appendix B.1. The major differences between the simple task and the Mandelbrot task will be discussed. This first important change comes with the class definition Figure 4.6. The class not only extends `PVMTask` but also implements `PVMMessageBufferListener` to handle the control messages. The `PVMMessageBufferListener` interface is how the object actually processes messages and is shown in Figure 4.7. When a message is received, the message tag is used to determine what needs to be done. The final difference between the Mandelbrot example and the simple example is the code to spawn off

```
try
{
    System.err.println("Exiting from PVM");
    task.exit();
}
catch (PVMException pvme)
{
    pvme.printStackTrace();
}
```

Figure 4.4. Exiting from PVM

worker tasks shown in Figure 4.8. Method overloading helps with code readability when there are no parameters or options that need to be sent in the `spawn()` call. This shows how little code is required to add PVM routines to a Java application.

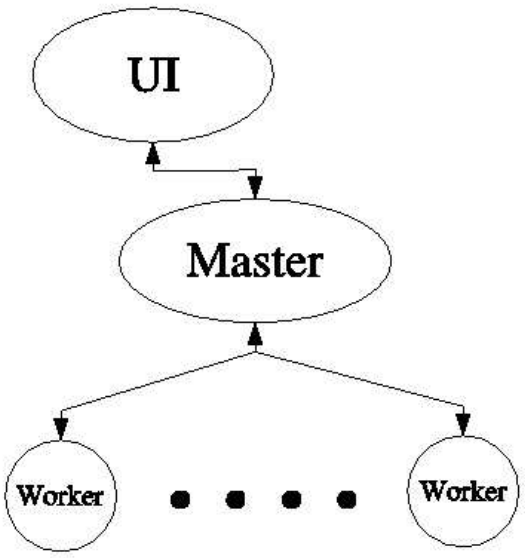

Figure 4.5. High Level System Overview of the Mandelbrot program

```
public class PVMMandelbrotMaster extends PVMTask
  implements PVMMessageBufferListener {
```

Figure 4.6. Class Definition PVMMandelbrotMaster

```
public void receivedPVMMessageBuffer(PVMMessageBuffer buffer) {
    try
    {
      int msgTag = buffer.getMessageTag();
      int tidFrom = buffer.getSenderTID();

      ...

      switch  (msgTag)
      {
        case INITIALDATAREQUEST:

          ...

          break;
        case WORKREQUEST:

          ...

          break;
        case WORKRESULT:

          ...

          break;
        case WORKEREXIT:

          ...

          break;
      }
    }
    catch (PVMException pvmex )
    {
      pvmex.printStackTrace();
    }
}
```

Figure 4.7. PVMMandelbrotMaster Message Listener Callback

```
int tids[] = new int[count];
PVM pvm = PVM.getInstance();
int infos = pvm.spawn("mandelbrot_worker",tids);
rtn = ( infos > 0 );
if ( !rtn ) {
    for ( int i = 0 ; i < tids.length; i++) {
      if ( tids[i] < 0 )
        System.err.println(i + ": "
              + PVM.pvm_errlist[-1 * tids[i]]);
    }
}
```

Figure 4.8. PVMMandelbrotMaster spawn code

Chapter 5

PERFORMANCE FINDINGS

To test the performance overhead that Java interjects into parallel programs, a couple of simple tests were run. Loop timing for simple arithmetic will show the raw performance difference between C and Java. Expecting Java would not win a flat out performance race with C, it was decided to test message throughput to see what performance penalty Java induced. Network communication is typically the bottleneck for parallel and distributed programs so message throughput is an important metric. Message throughput is tested two ways. One is with both tasks running on the same machine. PVM uses Unix sockets for communication on the same host.[1] This results in a low latency, high throughput channel. It is a good way to show the performance penalty that Java injects without having the network communication adversely affect the results. The other way throughput is measured is by passing messages between two separate hosts. This shows how typical network overhead affects the results.

5.1 Basic Loop Timing

Java is not known for being the fastest processing language. Doing simple arithmetic shows C's strengths. For example, Figures 5.1 and 5.2 show two simple programs

[1]Linux and many other Unix versions use Unix sockets, Windows using TCP/IP sockets

that just loop doing simple addition. Both were compiled with full optimizations and no debug informations to get the results in Table 5.1. C is almost three times faster than the Java equivalent.

```
int main(void) {
 long start = 0, stop = 0;
 int i = 0, loop = 0;
 start = GetTickCount();
 for ( loop = 0 ; loop < 100000000 ; loop++ )
 {
    i += loop;
 }
 stop = GetTickCount();
 fprintf(stderr," %ld\n",(stop-start));
}
```

Figure 5.1. Simple C Counter

TABLE 5.1 Simple Counter performance

	Timing
C	0.063s
Java	0.177s

5.2 Message Timing

Message throughput was measured by timing the time to send, receive, and verify various sizes of messages. Messages of size one, two, four, eight, sixteen, thirty-two, and sixty-four megabytes where used. Messages where sent in all combinations of Java programs and C programs and sending messages to the same host and to different hosts. The same hosts timing was done using a dual processor machine and the different hosts timing was done using two single processor machines. Send time is the

```
public class Counter
{
   public static void main(String[] args)
   {
      int i = 0;
      long start = System.currentTimeMillis();
      for ( int loop = 0; loop < 100000000 ; loop++)
      {
         i +=  loop;
      }

      long stop = System.currentTimeMillis();
      System.err.println(" " + (stop - start));
   }
}
```

Figure 5.2. Simple Java Counter

amount of time it takes to send the message. The receive timing is a combination of the receiving program receiving the message, re-sending it, and the sending program receiving the message. The validation step adds some processing to the timing by having the sending program validate that the message it sent matches the message it received. The total time metric was used for all comparisons. While Java cannot keep up with C for raw performance, once network traffic is included the difference becomes smaller. In Tables 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, and 5.8 comparing the percent speed up that C programs sending to C programs have over Java to Java programs shows that the speed up percentage decreases as the message size increases. The same host timing shows that there is penalty for using Java. A C to C program can be as high as 180% faster for small messages. A C program sending a one megabyte message to a C receiver on a different host is 35% faster than Java to Java com-

munication. For a 64MB message, however, the difference is only 20%. This can be particularly relevant for programs that access large databases or other data from across the network. Smaller sized messages show similar tendancies, with the difference widening slightly between Java and C. Communication overhead will drastically hurt any parallel program if too many small messages are being transmitted, however.

TABLE 5.2 Speed up Data 1MB Message

	Send (ms)	Receive (ms)	Verification (ms)	Total (ms)
	Percent Speedup	Percent Speedup	Percent Speedup	Percent Speedup
Java sender to Java receiver	10	43	2	56
same host	0%	0%	0%	0%
Java sender to C receiver	10	25	2	38
same host	0%	72%	0%	47%
C sender to Java receiver	8	29	1	38
same host	25%	48%	100%	47%
C sender to C receiver	8	11	1	20
same host	25%	291%	100%	180%
Java sender to Java receiver	11	133	2	147
different hosts	0%	0%	0%	0%
Java sender to C receiver	11	115	2	128
different hosts	0%	16%	0%	15%
C sender to Java receiver	8	117	1	127
different hosts	38%	14%	100%	16%
C sender to C receiver	8	100	1	109
different hosts	38%	33%	100%	35%

TABLE 5.3 Speed up Data 2MB Message

	Send (ms)	Receive (ms)	Verification (ms)	Total (ms)
	Percent Speedup	Percent Speedup	Percent Speedup	Percent Speedup
Java sender to Java receiver same host	23	68	3	95
	0%	0%	0%	0%
Java sender to C receiver same host	23	42	4	69
	0%	62%	-25%	38%
C sender to Java receiver same host	16	49	1	67
	43%	39%	200%	42%
C sender to C receiver same host	15	23	1	40
	53%	196%	200%	138%
Java sender to Java receiver different hosts	25	248	4	277
	0%	0%	0%	0%
Java sender to C receiver different hosts	24	221	3	250
	4%	12%	33%	11%
C sender to Java receiver different hosts	17	226	2	245
	47%	10%	100%	13%
C sender to C receiver different hosts	17	196	2	215
	47%	27%	100%	29%

5.3 Mandelbrot Timings

The Mandelbrot program was not used to provide timing comparisons. The purpose of it was to show that a PVM program could be written using Java for part of the program and use C to get higher performance. This combination illustrated what would be a typical PBj program with some parts in Java and others in C. No adverse affects were seen in the combination.

TABLE 5.4 Speed up Data 4MB Message

	Send (ms)	Receive (ms)	Verification (ms)	Total (ms)
	Percent Speedup	Percent Speedup	Percent Speedup	Percent Speedup
Java sender to Java receiver	47	120	7	175
same host	0%	0%	0%	0%
Java sender to C receiver	46	74	7	128
same host	2%	62%	0%	37%
C sender to Java receiver	32	89	3	125
same host	47%	35%	133%	40%
C sender to C receiver	32	46	3	125
same host	47%	161%	133%	116%
Java sender to Java receiver	50	477	7	535
different hosts	0%	0%	0%	0%
Java sender to C receiver	50	428	7	486
different hosts	0%	11%	0%	10%
C sender to Java receiver	33	446	3	483
different hosts	52%	7%	133%	11%
C sender to C receiver	33	394	3	432
different hosts	52%	21%	133%	24%

TABLE 5.5 Speed up Data 8MB Message

	Send (ms)	Receive (ms)	Verification (ms)	Total (ms)
	Percent Speedup	Percent Speedup	Percent Speedup	Percent Speedup
Java sender to Java receiver	96	218	15	331
same host	0%	0%	0%	0%
Java sender to C receiver	93	139	15	248
same host	3%	57%	0%	33%
C sender to Java receiver	66	169	7	242
same host	45%	29%	114%	37%
C sender to C receiver	65	91	7	163
same host	48%	140%	114%	103%
Java sender to Java receiver	100	934	15	1050
different hosts	0%	0%	0%	0%
Java sender to C receiver	99	839	15	954
different hosts	1%	11%	0%	10%
C sender to Java receiver	66	884	7	959
different hosts	52%	6%	114%	9%
C sender to C receiver	66	796	7	959
different hosts	52%	17%	114%	21%

TABLE 5.6 Speed up Data 16MB Message

	Send (ms)	Receive (ms)	Verification (ms)	Total (ms)
	Percent Speedup	Percent Speedup	Percent Speedup	Percent Speedup
Java sender to Java receiver	193	420	34	648
same host	0%	0%	0%	0%
Java sender to C receiver	186	266	34	486
same host	4%	58%	0%	22%
C sender to Java receiver	137	328	13	479
same host	41%	28%	162%	35%
C sender to C receiver	133	180	14	327
same host	45%	133%	143%	98%
Java sender to Java receiver	199	1829	36	2065
different hosts	0%	0%	0%	0%
Java sender to C receiver	197	1670	35	1902
different hosts	1%	10%	3%	9%
C sender to Java receiver	133	1736	13	1885
different hosts	50%	5%	177%	10%
C sender to C receiver	133	1568	18	1719
different hosts	50%	17%	100%	20%

TABLE 5.7 Speed up Data 32MB Message

	Send (ms)	Receive (ms)	Verification (ms)	Total (ms)
	Percent Speedup	Percent Speedup	Percent Speedup	Percent Speedup
Java sender to Java receiver	388	813	93	1297
same host	0%	0%	0%	0%
Java sender to C receiver	375	519	93	989
same host	3%	57%	0%	31%
C sender to Java receiver	277	642	28	951
same host	40%	27%	232%	36%
C sender to C receiver	262	354	27	644
same host	48%	130%	244%	101%
Java sender to Java receiver	389	3602	93	4087
different hosts	0%	0%	0%	0%
Java sender to C receiver	396	3298	93	3786
different hosts	-2%	9%	0%	8%
C sender to Java receiver	262	3446	32	3741
different hosts	48%	5%	191%	9%
C sender to C receiver	261	3147	32	3741
different hosts	49%	14%	191%	19%

TABLE 5.8 Speed up Data 64MB Message

	Send (ms)	Receive (ms)	Verification (ms)	Total (ms)
	Percent Speedup	Percent Speedup	Percent Speedup	Percent Speedup
Java sender to Java receiver	826	1661	206	2701
same host	0%	0%	0%	0%
Java sender to C receiver	802	1075	204	2083
same host	3%	55%	1%	30%
C sender to Java receiver	551	1268	54	1875
same host	50%	31%	281%	44%
C sender to C receiver	529	698	54	1282
same host	56%	138%	281%	111%
Java sender to Java receiver	799	7166	187	8159
different hosts	0%	0%	0%	0%
Java sender to C receiver	780	6611	187	7589
different hosts	2%	8%	0%	8%
C sender to Java receiver	528	6829	63	7418
different hosts	51%	5%	197%	10%
C sender to C receiver	535	6243	57	6834
different hosts	49%	15%	228%	19%

Chapter 6

PROBLEMS/ISSUES

6.1 Java Induced Issues

Java behaves differently than C so that are some differences in PVM programs written in C versus those written in Java. `PvmDataInPlace` cannot be used due to Java's memory model and the use of JNI. This can provide some performance penalties from copying data mutliple times. Another Java requirement is that objects can only be sent using the encoding type `PvmDataRaw`. This really is not a problem since the only task that would receive the object would also be a Java task. Java defines the endianess for the run-time so no ill-transitions will occur.

6.2 JNI Layer

Using JNI presents a variety of problems with Java programs. Executing code outside of the JVM introduces another layer of complexity and also the possibility of program crashes. The concept of the JVM class loader also has problems with JNI access from classes loaded from different classloaders. For instance, the popular Servlet container Tomcat [6] uses multiple class loaders to isolate the various web-apps that are being served. Loading the JNI library through one of these auxillary class loaders creates

stability issues that were discovered when developing a simple servlet to show tasks running in the virtual machine. Many recommended work-arounds failed to solve the problem.

6.3 Performance

In the simplest of problems, C outshines Java's performance. A simple loop and count for integers shows a nearly three-fold speed benefit for C. Once processing becomes more complex, however, the performance gains are not as great. Sun's own cryptography library was implemented in Java instead of native code because it actually performed better. [14]

Chapter 7

CONCLUSIONS

PVM bindings in Java provides a useful tool for developing parallel programs. Not only can Java's features be leveraged to provide new uses for parallel programs, but exising PVM programs can be enhanced via Java extensions (for instance by adding a GUI.) Future enhancements to PBJ could include re-writing the underlying JNI calls to use straight Java but as it stands, PBJ is ready to be used for non-trivial programs. Another possible enhancement would be to take the work done to link Java and PVM and extend the same ideas and principles to MPI. This would provide two communications primitives for parallel programs written in Java. Even though Java cannot keep up with C for raw performance, other factors weigh into making a tool useful for writing parallel programs.

REFERENCES

[1] Thomas Sterling. *How To Build a Beowulf.* MIT Press, USA, 1998.

[2] Cay Horstman and Gary Cornell *Core Java Volume 1.* Prentice Hall, USA, 1999.

[3] PVM: Parallel Virtual Machine. http://www.csm.ornl.gov/pvm/pvm_home.html

[4] jPVM: A native methods interface to PVM for the Java platform http://www.chmsr.gatech.edu/jPVM/

[5] The JPVM Home Page http://www.cs.virginia.edu/ãjf2j/jpvm.html

[6] The Jakarta Site - Apache Tomcat. http://jakarta.apache.org/tomcat/index.html

[7] Red Hat. http://www.redhat.com

[8] Microsoft. http://www.microsoft.com

[9] Apache Ant. http://ant.apache.org

[10] Junit Test Framework. http://junit.org

[11] Log4J Logging Framework. http://jakarta.apache.org/log4j

[12] Java Developer Site. http://java.sun.com

[13] Barry Wilkinson and Michael Allen. *Parallel Programming.* Prentice Hall, USA, 1999.

[14] Cay Horstman and Gary Cornell. *Core Java Volume 2.* Prentice Hall, USA, 2000.

[15] The Globus Alliance http://www.globus.org

Appendix A

TIMING CODE

A.1 Java Timing Task

```
package samples.timing;

import PBJ.*;


public class TimingTask extends PVMTask
{
    public static final int SIZE = 1000;
    public static final int LOOP = 1000000000;

    public static void main(String[] args)
    {
        int data[] = null;
        int toTID = -1;

        if (args.length > 0)
        {
            toTID = Integer.parseInt(args[0]);
        }

        TimingTask task = new TimingTask();

        //
        //  Enroll in PVM
        //
        try
        {
            System.err.println("My tid: " + task.myTID());
        }
```

```
catch (PVMException pvme)
{
    pvme.printStackTrace();
    System.exit(1);
}

PVMMessageBuffer buffer = null;
PVMMessageBuffer sender = null;


for ( int l = 0; l < LOOP; l++)
{
    sender = null;
    buffer = null;
    data = null;


    // Look to see if task is the receiver or sender
    if (toTID < 0)
    {

        // receiver....
        try
        {
            sender = PVMMessageBuffer.makeBuffer(PvmDataRaw);
            System.err.println("Waiting...." + l);
            // receive
            buffer = task.receive(-1, 1);
            data = buffer.unPackInt(SIZE);

            // return
            sender.pack(data);
            sender.send(buffer.getSenderTID(),1);
            System.err.println("Received " + data.length
                               + " integers");
            buffer.freeBuffer();
            sender.freeBuffer();
        }
        catch (PVMException pvme)
        {
            pvme.printStackTrace();
        }
    }
    else
    {
```

```
// sender
try
{
    sender = PVMMessageBuffer.makeBuffer(PvmDataRaw);
    data = new int[SIZE];

    // build up what we're sending
    for ( int i = 0 ; i < SIZE ; i++)
    {
        data[i] = i;
    }
    System.out.print(l + ",");
    long start = System.currentTimeMillis();
    sender.pack(data);

    // send it off
    sender.send(toTID, 1);

    long sent = System.currentTimeMillis();
    // wait for the rebound
    buffer = task.receive(-1, 1);
    int rtn[] = buffer.unPackInt(SIZE);

    long rxed = System.currentTimeMillis();
    for ( int i = 0 ; i < SIZE ; i++)
    {
        if ( rtn[i] != data[i] )
        {
          System.err.println("Data mismatch at index "
                             + i);
        }
    }
    buffer.freeBuffer();
    sender.freeBuffer();
    long stop = System.currentTimeMillis();
    System.out.print((sent - start) + ",");
    System.out.print((rxed - sent) + ",");
    System.out.print((stop - rxed) + ",");
    System.out.println((stop - start));
}
catch (PVMException pvme)
{
    pvme.printStackTrace();
}
```

```
            }
        } // end of LOOP

        //
        // Exit from PVM
        //
        try
        {
            System.err.println("Exiting from PVM");
            task.exit();
        }
        catch (PVMException pvme)
        {
            pvme.printStackTrace();
        }
    }
}
```

A.2 C Timing Task

```
#include <stdio.h>
#include <pvm3.h>
#include "timing.h"

#define SIZE 1000000
#define LOOPS 1000

void CheckReturn( int rtn )
{
  if ( rtn < PvmOk ) {
    pvm_perror( "Fatal Error" );
    exit( 1 );
  }
}

int main( int argc, char * * argv )
{
  int myTid = -1;
  int toTid = -1;
  int info = 0;
  int list1[SIZE];
```

```
int list2[SIZE];
int loop1, loop2 = 0;
int bufID = 0, bytes = 0, msgTag = 0, sendTID = 0;
long start = 0, stop = 0, recv = 0, sent = 0;

if ( argc > 1 )
  toTid = atoi( argv[1] );


myTid = pvm_mytid();
CheckReturn( myTid );

fprintf( stderr, "My tid: %d\n", myTid );

for ( loop2 = 0; loop2 < LOOPS; loop2++ ) {
  if ( toTid < 0 ) {
    /* receiver */

    fprintf( stderr, "Waiting...%d\n", loop2 );
    bufID = pvm_recv( -1, 1 );
    CheckReturn( bufID );

    info = pvm_upkint( list1, SIZE, 1 );
    CheckReturn( info );

    info = pvm_bufinfo( bufID, & bytes, & msgTag, & sendTID );
    CheckReturn( info );

    bufID = pvm_initsend( PvmDataRaw );
    CheckReturn( bufID );

    info = pvm_pkint( list1, SIZE, 1 );
    CheckReturn( info );

    info = pvm_send( sendTID, 1 );
    CheckReturn( info );

    fprintf( stderr, "Received %d integers from %d\n",
                      bytes / sizeof( int ), sendTID );

  } else {

    bufID = pvm_initsend( PvmDataRaw );
    CheckReturn( bufID );
    /* sender */
```

```
      for ( loop1 = 0; loop1 < SIZE; loop1++ ) {
        list1[loop1] = loop1;
      }

      printf( "%d,", loop2 );
      start = GetTickCount();

      info = pvm_pkint( list1, SIZE, 1 );
      CheckReturn( info );

      info = pvm_send( toTid, 1 );
      CheckReturn( info );
      sent = GetTickCount();
      info = pvm_recv( -1, 1 );
      info = pvm_upkint( list2, SIZE, 1 );
      CheckReturn( info );

      recv = GetTickCount();

      for ( loop1 = 0; loop1 < SIZE; loop1++ ) {
        if ( list1[loop1] != list2[loop1] ) {
          fprintf( stderr, "Data mismatch at index %d\n", loop1 );
        }
      }
      stop = GetTickCount();

      printf( "%d,%d,%d,%d\n", ( sent - start ), ( recv - sent ),
                               ( stop - recv ), ( stop - start ) );

    }
  }
  fprintf( stderr, "Exiting from PVM\n" );
  info = pvm_exit();
  CheckReturn( info );

  return 0;
}
```

Appendix B

MANDELBROT PVM CODE

B.1 Java code

```
package samples.mandlebrot;

import PBJ.*;

public class PVMMandelbrotMaster extends PVMTask
    implements PVMMessageBufferListener {

    Mandelbrot mb;
    int count;

    protected static final int INITIALDATAREQUEST = 1;
    protected static final int INITIALDATARESPONSE = 2;
    protected static final int WORKREQUEST = 3;
    protected static final int WORKRESPONSE = 4;
    protected static final int WORKRESULT = 5;
    protected static final int ENDWORKER = 6;
    protected static final int WORKEREXIT = 7;
    protected static final int PARENTEXIT = 8;


    public PVMMandelbrotMaster(Mandelbrot _mb)
    {
      mb = _mb;
      addPVMMessageBufferListener(this);
    }


    public void receivedPVMMessageBuffer(PVMMessageBuffer buffer) {
        try
```

```
{
  int msgTag = buffer.getMessageTag();
  int tidFrom = buffer.getSenderTID();

  PVMMessageBuffer sendBuffer = PVMMessageBuffer.makeBuffer();

  switch  (msgTag)
  {
    case INITIALDATAREQUEST:
      count++;
      sendBuffer.pack(Mandelbrot.getRealMin());
      sendBuffer.pack(Mandelbrot.getRealMax());
      sendBuffer.pack(Mandelbrot.getImaginaryMin());
      sendBuffer.pack(Mandelbrot.getImaginaryMax());
      sendBuffer.pack(mb.getDisplayWidth());
      sendBuffer.pack(mb.getDisplayHeight());
      sendBuffer.pack(mb.getMax());
      sendBuffer.send(tidFrom, INITIALDATARESPONSE);
       break;
    case WORKREQUEST:
      int next = mb.getNextIter();
      sendBuffer.pack(next);
      sendBuffer.send(tidFrom, WORKRESPONSE);
      break;
    case WORKRESULT:
      int col = buffer.unPackInt();
      int data[] = buffer.unPackInt(mb.getDisplayHeight());
      mb.drawColumn(col, data);
      break;
    case WORKEREXIT:
      int workDone = buffer.unPackInt();
      System.out.println("Worker: " + tidFrom + " did "
                         + workDone + " work units");
      if ( --count == 0 )
      {
          mb.endit();
          disableMessageEvents();
      }
      break;
  }
}
catch (PVMException pvmex )
{
  pvmex.printStackTrace();
}
```

```
    }

    public boolean spawn(int count)
    {
        enableMessageEvents(100);
        boolean rtn = true;
        try
        {
            int tids[] = new int[count];
            PVM pvm = PVM.getInstance();
            int infos = pvm.spawn("mandelbrot_worker",tids);
            rtn = ( infos > 0 );
            if ( !rtn ) {
                for ( int i = 0 ; i < tids.length; i++) {
                  if ( tids[i] < 0 )
                    System.err.println(i + ": "
                          + PVM.pvm_errlist[-1 * tids[i]]);
                }
            }
        }
        catch ( PVMException pvmex )
        {
            pvmex.printStackTrace();
            rtn = false;
        }
        return rtn;
    }

    public int lookingForTID() {
        return -1;
    }

    public int lookingForMessageTag() {
        return -1;
    }

}
```

B.2 C code

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <pvm3.h>

#include <sys/time.h>
/*
 * gcc -L/usr/share/pvm3/lib/LINUXI386/ -I/usr/share/pvm3/include/ \
 *  -o mandelbrot_worker mandelbrot_worker.c -lpvm3
 */

/*
 *  While these would normally go in a .h file, there is no "sharing"
 *  between Java and C so we just stuff them here
 */
#define INITIALDATAREQUEST         1
#define INITIALDATARESPONSE        2
#define WORKREQUEST                3
#define WORKRESPONSE               4
#define WORKRESULT                 5
#define ENDWORKER                  6
#define WORKEREXIT                 7
#define PARENTEXIT                 8

void checkPVMError(int result, char *msg)
{
    //printf("Checking result from %s\n", msg);
    if ( result < PvmOk ) {
        pvm_perror(msg);
        exit(1);
    }
}

void checkParentExit(int parentTID) {
    int info = pvm_probe ( parentTID, PARENTEXIT );
    if ( info > 0 ) {
        printf("Parent exited.  Exiting.\n");
        pvm_exit();
        exit(1);
    }
}

// does the actual calculation for a pixel
int calculatePixel(float c_real, float c_imaginary, int max)
{
    int count = 0;
```

```
    float temp = 0.0F, lengthSq = 0.0F;
    float z_real = 0.0F;
    float z_imaginary = 0.0F;

    do {
        temp = (z_real * z_real) -
               (z_imaginary * z_imaginary) + c_real;
        z_imaginary = (2 * z_real * z_imaginary) + c_imaginary;
        z_real = temp;
        lengthSq = z_real * z_real + z_imaginary * z_imaginary;
    } while ( (lengthSq < 4.0F) && (++count < max) );

    return count;
}


int main(int argc, char **argv)
{
    // things to fetch before starting
    float realMin = 0.0F;
    float realMax = 0.0F;
    float imaginaryMin = 0.0F;
    float imaginaryMax = 0.0F;
    int displayWidth = 0;
    int displayHeight = 0;
    int max = 0;

    //  Lets us know when we are done
    int more = 1;


    int myParent = pvm_parent();
    checkPVMError(myParent, "pvm_parent");

    int tids[1];
    tids[0] = myParent;
    int info = pvm_notify( PvmTaskExit, PARENTEXIT, 1, tids );

    // time out so we don't just wait for ever
    struct timeval tmout;
    tmout.tv_sec = 60;
    tmout.tv_usec = 0;

    // request inital data
    info = pvm_initsend(PvmDataDefault);
```

```
checkPVMError(info,"initsend - ask for initial data");
info = pvm_send(myParent,INITIALDATAREQUEST);
checkPVMError(info,"send - ask for initial data");

checkParentExit(myParent);

// receive the inital data
int bufid = pvm_trecv(myParent,INITIALDATARESPONSE,&tmout);
checkPVMError(bufid,"recv - initial data");
info = pvm_upkfloat(&realMin,1,1);
checkPVMError(info,"pvm_upkfloat - realMin");
info = pvm_upkfloat(&realMax,1,1);
checkPVMError(info,"pvm_upkfloat - realMax");
info = pvm_upkfloat(&imaginaryMin,1,1);
checkPVMError(info,"pvm_upkfloat - imaginaryMin");
info = pvm_upkfloat(&imaginaryMax,1,1);
checkPVMError(info,"pvm_upkfloat - imaginaryMax");
info = pvm_upkint(&displayWidth,1,1);
checkPVMError(info,"pvm_upkint - displayWidth");
info = pvm_upkint(&displayHeight,1,1);
checkPVMError(info,"pvm_upkint - displayHeight");
info = pvm_upkint(&max,1,1);
checkPVMError(info,"pvm_upkint - max");

float scaleReal = (realMax - realMin)/displayWidth;
float scaleImaginary =
                (imaginaryMax - imaginaryMin)/displayHeight;

int workdone = 0;
do {
    float c_real = 0.0F;
    float c_imaginary = 0.0F;


    // request a round of work....
    int info = pvm_initsend(PvmDataDefault);
    checkPVMError(info,"initsend - ask for work");
    info = pvm_send(myParent,WORKREQUEST);
    checkPVMError(info,"send - ask for work");

    checkParentExit(myParent);

    // receive a round of work....
tmout.tv_sec = 60;
tmout.tv_usec = 0;
```

```
    int bufid = pvm_trecv(myParent,WORKRESPONSE,&tmout);
    checkPVMError(bufid,"recv - work");

    // get the work
    int column = -1;
    info = pvm_upkint(&column,1,1);
    checkPVMError(info,"pvm_upkint - column");

    // do the work....
    if ( column < 0 ) {
        more = 0;
        printf("Done with work...\n");
        continue;
    }

    workdone++;
    int vert[displayHeight];
    int y = 0;
    for ( y = 0; y < displayHeight ; y++ )
    {
        c_real = realMin + ((float)column * scaleReal);
        c_imaginary = imaginaryMin + ((float)y * scaleImaginary);
        vert[y] = calculatePixel(c_real, c_imaginary,max);
    }

    // send the results
    info = pvm_initsend(PvmDataDefault);
    checkPVMError(info,"initsend");

    info = pvm_pkint(&column,1,1);
    checkPVMError(info,"pkint");


    info = pvm_pkint(vert,displayHeight,1);
    checkPVMError(info,"pkint");

    info = pvm_send(myParent,WORKRESULT);
    checkPVMError(info,"send");

    checkParentExit(myParent);

} while ( more );

    info = pvm_initsend(PvmDataDefault);
    checkPVMError(info,"initsend");
```

```
        info = pvm_pkint(&workdone,1,1);
        checkPVMError(info,"pkint");

        info = pvm_send(myParent,WORKEREXIT);
        checkPVMError(info,"send");

    printf("Exit...\n");
    return pvm_exit();
}  // end of main....
```

Appendix C

PERFORMANCE DATA

Values obtained from the `Beowulf` cluster maintained by the Computer Science Department at Boise State University and funded by NSF Grant 0321233. The head node is a dual 2.4 GHz Xeon processor Linux system with 4 GB of RAM running a 2.4.24 kernel. The nodes are dual 2.4 GHz Xeon processors with 1 GB of RAM. The heap size for Java was increased to 256 megabytes so that memory allocation would not be a problem. Tests where run ten times. Results where paired down to eight by removing minimum and maximum values. The Java programs where compiled using Java 1.4.2 turning off debug options (-g:none). The C programs where compiled using GCC 3.2.2 with -O3 optimizations turned on. Timing in Java used `System.currentTimeMillis()` . The C timing used some custom subroutines based on `gettimeofday()`.

C.1 Loop Timing

TABLE C.1 Java Loop Timings

Time (seconds)
.217
.184
.207
.168
.168
.168
.168
.184
.168
.172
.177

TABLE C.2 C Loop Timings

Time (seconds)
.062
.063
.062
.063
.063
.062
.063
.063
.063
.063
.063

C.2 Message Timing

TABLE C.3 C Sender to C Receiver, 1 Megabyte Message, Same Host

Send	Receive	Verify	Total
9	13	1	23
8	11	1	20
8	11	1	20
8	11	1	20
8	11	1	20
8	12	0	20
8	12	1	21
8	12	1	21
7	12	1	20
8	12	1	21
8	11	1	20

TABLE C.4 C Sender to C Receiver, 1 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
9	102	1	112
8	101	1	110
8	101	1	110
9	100	1	110
9	101	1	111
8	100	1	109
8	98	2	108
8	99	1	108
9	100	1	110
9	99	1	109
8	100	1	109

TABLE C.5 C Sender to C Receiver, 2 Megabyte Message, Same Host

Send	Receive	Verify	Total
18	34	2	54
15	23	2	40
16	23	2	41
16	23	2	41
15	24	1	40
16	23	2	41
15	23	2	40
16	23	1	40
16	23	2	41
16	23	2	41
15	23	1	40

TABLE C.6 C Sender to C Receiver, 2 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
19	202	2	223
16	201	2	219
17	200	2	219
17	196	2	215
17	193	2	212
17	193	2	212
18	192	2	212
17	197	2	216
17	197	2	216
17	198	2	217
17	196	2	215

TABLE C.7 C Sender to C Receiver, 4 Megabyte Message, Same Host

Send	Receive	Verify	Total
37	51	3	91
32	46	4	82
32	45	4	81
32	46	4	82
33	46	3	82
32	46	3	81
32	45	4	81
32	46	3	81
32	47	3	82
32	48	3	83
32	46	3	81

TABLE C.8 C Sender to C Receiver, 4 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
37	398	3	438
34	385	4	423
33	400	3	436
33	402	3	438
33	402	3	438
33	400	4	437
34	390	4	428
34	388	3	425
34	388	3	425
34	392	3	429
33	394	3	432

TABLE C.9 C Sender to C Receiver, 8 Megabyte Message, Same Host

Send	Receive	Verify	Total
76	109	7	192
65	92	6	163
66	91	7	164
65	91	7	163
65	92	7	164
66	91	7	164
64	90	7	161
66	90	7	163
66	90	7	163
65	93	7	165
65	91	7	163

TABLE C.10 C Sender to C Receiver, 8 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
75	808	7	890
66	793	7	866
66	797	8	871
66	789	8	863
65	799	8	872
68	781	8	857
66	792	8	866
65	799	8	872
65	800	8	873
66	800	8	874
66	796	7	869

TABLE C.11 C Sender to C Receiver, 16 Megabyte Message, Same Host

Send	Receive	Verify	Total
148	203	14	365
131	180	14	325
132	179	14	325
133	181	13	327
133	181	14	328
133	181	14	328
133	178	14	325
134	180	14	328
134	180	14	328
132	184	14	330
133	180	14	327

TABLE C.12 C Sender to C Receiver, 16 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
147	1633	18	1798
135	1543	18	1696
132	1562	18	1712
135	1551	18	1704
131	1599	18	1748
135	1574	18	1727
131	1576	18	1725
133	1570	18	1721
131	1575	18	1724
135	1540	18	1693
133	1568	18	1719

TABLE C.13 C Sender to C Receiver, 32 Megabyte Message, Same Host

Send	Receive	Verify	Total
279	435	27	741
260	353	28	641
260	352	28	640
260	354	27	641
265	353	28	646
263	354	27	644
262	356	27	645
262	354	27	643
264	355	27	646
262	358	28	648
262	354	27	644

TABLE C.14 C Sender to C Receiver, 32 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
293	3156	33	3482
262	3126	33	3421
264	3137	32	3433
261	3168	32	3461
261	3173	33	3467
260	3137	33	3430
261	3130	33	3424
260	3156	33	3449
262	3114	33	3409
260	3181	33	3474
261	3147	32	3444

TABLE C.15 C Sender to C Receiver, 64 Megabyte Message, Same Host

Send	Receive	Verify	Total
600	808	54	1462
533	701	55	1289
527	694	55	1276
528	697	55	1280
529	695	55	1279
531	701	55	1287
530	695	54	1279
528	693	55	1276
528	700	55	1283
532	701	55	1288
529	698	54	1282

TABLE C.16 C Sender to C Receiver, 64 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
589	6420	58	7067
524	6333	57	6914
542	6126	56	6724
522	6368	57	6947
533	6243	57	6833
542	6138	57	6737
541	6141	57	6739
542	6211	57	6810
541	6241	57	6839
522	6273	58	6853
535	6243	57	6834

TABLE C.17 C Sender to Java Receiver, 1 Megabyte Message, Same Host

Send	Receive	Verify	Total
9	37	1	47
8	29	1	38
8	29	1	38
7	31	1	39
8	29	1	38
8	29	1	38
8	29	1	38
8	29	1	38
8	29	1	38
8	29	1	38
8	29	1	38

TABLE C.18 C Sender to Java Receiver, 1 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
10	123	1	134
9	116	1	126
9	117	1	127
9	118	1	128
9	117	1	127
9	118	1	128
8	118	1	127
8	116	1	125
9	117	1	127
9	117	1	127
8	117	1	127

TABLE C.19 C Sender to Java Receiver, 2 Megabyte Message, Same Host

Send	Receive	Verify	Total
18	63	1	82
16	49	1	66
16	49	1	66
16	50	2	68
16	50	2	68
16	50	2	68
16	49	2	67
16	50	1	67
16	49	2	67
16	49	2	67
16	49	1	67

TABLE C.20 C Sender to Java Receiver, 2 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
19	235	2	256
17	226	2	245
16	229	2	247
17	228	2	247
17	227	2	246
17	225	2	244
17	225	2	244
17	225	2	244
17	225	2	244
17	230	2	249
17	226	2	245

TABLE C.21 C Sender to Java Receiver, 4 Megabyte Message, Same Host

Send	Receive	Verify	Total
37	111	4	152
33	86	4	123
32	87	4	123
33	88	3	124
32	89	4	125
32	89	4	125
32	90	4	126
32	90	4	126
32	90	3	125
33	90	5	128
32	89	3	125

TABLE C.22 C Sender to Java Receiver, 4 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
38	454	4	496
33	442	4	479
35	434	3	472
33	451	3	487
33	445	4	482
34	447	3	484
34	439	4	477
34	448	4	486
33	449	4	486
33	451	4	488
33	446	3	483

TABLE C.23 C Sender to Java Receiver, 8 Megabyte Message, Same Host

Send	Receive	Verify	Total
76	208	7	291
66	163	7	236
66	162	7	235
66	167	7	240
66	170	7	243
65	171	7	243
66	169	7	242
66	170	7	243
66	171	7	244
66	171	11	248
66	169	7	242

TABLE C.24 C Sender to Java Receiver, 8 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
77	884	7	968
67	874	8	949
66	885	7	958
65	883	7	955
65	883	7	955
65	896	8	969
68	871	8	947
66	894	8	968
65	883	8	956
66	890	8	964
66	884	7	959

TABLE C.25 C Sender to Java Receiver, 16 Megabyte Message, Same Host

Send	Receive	Verify	Total
154	385	13	552
137	317	14	468
137	317	14	468
136	320	13	469
138	333	14	485
139	332	14	485
137	333	14	484
137	334	13	484
138	330	14	482
136	330	14	480
137	328	13	479

TABLE C.26 C Sender to Java Receiver, 16 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
151	1746	13	1910
132	1747	13	1892
136	1701	13	1850
131	1738	13	1882
132	1744	14	1890
131	1757	13	1901
137	1704	13	1854
136	1725	13	1874
132	1734	13	1879
133	1769	13	1915
133	1736	13	1885

TABLE C.27 C Sender to Java Receiver, 32 Megabyte Message, Same Host

Send	Receive	Verify	Total
318	685	28	1031
280	616	27	923
276	621	28	925
278	616	55	949
277	648	29	954
275	651	28	954
275	647	28	950
274	653	28	955
278	651	28	957
277	654	33	964
277	642	28	951

TABLE C.28 C Sender to Java Receiver, 32 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
302	3479	33	3814
262	3411	32	3705
260	3404	32	3696
260	3435	32	3727
260	3445	33	3738
259	3474	32	3765
259	3452	32	3743
260	3491	32	3783
269	3390	32	3691
268	3475	33	3776
262	3446	32	3741

TABLE C.29 C Sender to Java Receiver, 64 Megabyte Message, Same Host

Send	Receive	Verify	Total
620	1498	54	2172
547	1216	55	1818
547	1220	54	1821
554	1208	54	1816
553	1287	54	1894
552	1284	54	1890
549	1282	55	1886
551	1285	55	1891
555	1286	55	1896
553	1287	69	1909
551	1268	54	1875

TABLE C.30 C Sender to Java Receiver, 64 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
606	6912	63	7581
537	6826	62	7425
537	6745	63	7345
537	6699	63	7299
523	6885	64	7472
522	6866	63	7451
521	6794	63	7378
536	6804	63	7403
517	6862	63	7442
515	6856	63	7434
528	6829	63	7418

TABLE C.31 Java Sender to C Receiver, 1 Megabyte Message, Same Host

Send	Receive	Verify	Total
14	31	2	47
11	27	2	40
10	25	2	37
11	26	2	39
11	25	2	38
11	25	2	38
10	26	1	37
10	26	2	38
10	25	2	37
11	27	2	40
10	25	2	38

TABLE C.32 Java Sender to C Receiver, 1 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
14	115	2	131
12	117	1	130
12	115	2	129
11	114	3	128
12	114	2	128
11	115	2	128
12	116	2	130
11	116	2	129
11	115	2	128
12	113	2	127
11	115	2	128

TABLE C.33 Java Sender to C Receiver, 2 Megabyte Message, Same Host

Send	Receive	Verify	Total
26	45	4	75
23	45	4	72
23	41	4	68
23	43	4	70
23	41	4	68
24	41	4	69
23	42	3	68
22	42	4	68
22	42	4	68
23	42	4	69
23	42	4	69

TABLE C.34 Java Sender to C Receiver, 2 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
27	223	4	254
25	224	4	253
25	217	4	246
24	224	3	251
25	218	3	246
25	221	4	250
24	223	4	251
25	223	4	252
25	222	4	251
25	217	4	246
24	221	3	250

TABLE C.35 Java Sender to C Receiver, 4 Megabyte Message, Same Host

Send	Receive	Verify	Total
52	90	8	150
46	78	8	132
46	73	8	127
46	77	8	131
46	74	8	128
46	74	7	127
46	74	7	127
46	74	8	128
46	73	7	126
46	74	7	127
46	74	7	128

TABLE C.36 Java Sender to C Receiver, 4 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
56	437	8	501
51	435	8	494
50	434	8	492
50	431	7	488
51	416	8	475
50	425	8	483
49	430	7	486
51	426	8	485
50	427	7	484
51	421	7	479
50	428	7	486

TABLE C.37 Java Sender to C Receiver, 8 Megabyte Message, Same Host

Send	Receive	Verify	Total
107	175	16	298
92	146	16	254
93	137	15	245
93	145	15	253
93	137	16	246
94	137	15	246
93	138	15	246
94	137	16	247
94	137	15	246
94	140	15	249
93	139	15	248

TABLE C.38 Java Sender to C Receiver, 8 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
107	870	16	993
101	847	15	963
100	838	15	953
100	832	16	948
98	826	15	939
97	849	15	961
100	840	15	955
100	829	15	944
97	841	16	954
99	843	16	958
99	839	15	954

TABLE C.39 Java Sender to C Receiver, 16 Megabyte Message, Same Host

Send	Receive	Verify	Total
205	297	34	536
185	280	34	499
186	265	34	485
185	277	35	497
186	260	35	481
184	260	34	478
186	262	35	483
186	259	35	480
187	262	35	484
187	263	35	485
186	266	34	486

TABLE C.40 Java Sender to C Receiver, 16 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
214	1725	35	1974
197	1693	36	1926
198	1660	35	1893
197	1681	36	1914
198	1644	35	1877
198	1663	35	1896
197	1659	35	1891
197	1663	36	1896
196	1682	35	1913
192	1661	35	1888
197	1670	35	1902

TABLE C.41 Java Sender to C Receiver, 32 Megabyte Message, Same Host

Send	Receive	Verify	Total
400	646	64	1110
377	511	108	996
375	510	108	993
375	533	68	976
378	512	108	998
376	509	108	993
377	534	69	980
374	506	108	988
373	509	108	990
375	538	68	981
375	519	93	989

TABLE C.42 Java Sender to C Receiver, 32 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
421	3435	69	3925
405	3245	107	3757
391	3308	107	3806
391	3300	69	3760
398	3246	109	3753
395	3254	108	3757
399	3314	69	3782
400	3298	107	3805
389	3292	108	3789
391	3375	69	3835
396	3298	93	3786

TABLE C.43 Java Sender to C Receiver, 64 Megabyte Message, Same Host

Send	Receive	Verify	Total
855	1234	137	2226
739	1012	217	1968
741	1015	218	1974
812	1138	162	2112
818	1029	240	2087
809	1075	240	2124
807	1144	161	2112
812	1031	240	2083
811	1004	241	2056
810	1156	155	2121
802	1075	204	2083

TABLE C.44 Java Sender to C Receiver, 64 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
860	6670	139	7669
783	6650	215	7648
786	6588	216	7590
771	6690	139	7600
786	6652	217	7655
790	6562	217	7569
770	6605	140	7515
772	6581	217	7570
771	6577	217	7565
783	6572	140	7495
780	6611	187	7589

TABLE C.45 Java Sender to Java Receiver, 1 Megabyte Message, Same Host

Send	Receive	Verify	Total
13	52	2	67
10	46	2	58
11	43	2	56
10	46	2	58
10	43	2	55
10	43	2	55
10	43	2	55
11	43	2	56
11	43	1	55
10	43	2	55
10	43	2	56

TABLE C.46 Java Sender to Java Receiver, 1 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
14	141	2	157
11	137	2	150
11	134	2	147
12	134	2	148
12	133	2	147
12	134	2	148
11	133	2	146
12	133	2	147
12	132	2	146
12	132	1	145
11	133	2	147

TABLE C.47 Java Sender to Java Receiver, 2 Megabyte Message, Same Host

Send	Receive	Verify	Total
26	82	4	112
24	71	4	99
23	67	4	94
23	71	3	97
23	68	4	95
23	68	4	95
24	68	3	95
23	68	4	95
24	68	3	95
23	67	4	94
23	68	3	95

TABLE C.48 Java Sender to Java Receiver, 2 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
28	257	4	289
25	255	4	284
24	251	4	279
25	250	4	279
26	245	4	275
25	245	4	274
25	247	4	276
24	247	4	275
26	247	4	277
25	247	4	276
25	248	4	277

TABLE C.49 Java Sender to Java Receiver, 4 Megabyte Message, Same Host

Send	Receive	Verify	Total
52	142	7	201
47	123	7	177
47	118	7	172
48	123	8	179
47	120	7	174
46	120	8	174
47	119	8	174
47	120	7	174
47	118	7	172
47	120	10	177
47	120	7	175

TABLE C.50 Java Sender to Java Receiver, 4 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
55	488	8	551
51	486	8	545
50	473	8	531
49	479	8	536
52	468	8	528
51	474	7	532
50	475	8	533
51	470	7	528
50	478	7	535
50	483	7	540
50	477	7	535

TABLE C.51 Java Sender to Java Receiver, 8 Megabyte Message, Same Host

Send	Receive	Verify	Total
101	237	16	354
95	218	16	329
95	210	15	320
95	223	15	333
96	218	15	329
96	219	15	330
97	217	15	329
97	218	15	330
97	217	16	330
97	221	24	342
96	218	15	331

TABLE C.52 Java Sender to Java Receiver, 8 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
107	962	15	1084
99	944	16	1059
100	913	15	1028
102	937	15	1054
100	924	16	1040
98	948	15	1061
101	931	16	1048
99	933	16	1048
101	932	15	1048
102	930	16	1048
100	934	15	1050

TABLE C.53 Java Sender to Java Receiver, 16 Megabyte Message, Same Host

Send	Receive	Verify	Total
210	438	34	682
194	422	34	650
193	400	35	628
190	420	34	644
194	421	34	649
195	418	35	648
192	421	35	648
194	415	35	644
193	419	34	646
192	429	36	657
193	420	34	648

TABLE C.54: Java Sender to Java Receiver, 16 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
219	1876	36	2131
200	1814	35	2049
199	1811	36	2046
202	1835	36	2073
197	1835	36	2068
200	1828	36	2064
201	1811	36	2048
200	1815	36	2051
196	1861	36	2093
200	1838	36	2074
199	1829	36	2065

TABLE C.55 Java Sender to Java Receiver, 32 Megabyte Message, Same Host

Send	Receive	Verify	Total
427	959	68	1454
387	785	108	1280
391	785	108	1284
387	802	69	1258
385	811	108	1304
389	808	108	1305
391	835	69	1295
387	809	108	1304
388	806	109	1303
388	852	68	1308
388	813	93	1297

TABLE C.56: Java Sender to Java Receiver, 32 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
434	3672	70	4176
392	3560	108	4060
382	3611	108	4101
395	3580	70	4045
383	3653	108	4144
391	3568	108	4067
388	3579	70	4037
383	3646	109	4138
393	3570	109	4072
388	3616	70	4074
389	3602	93	4087

TABLE C.57 Java Sender to Java Receiver, 64 Megabyte Message, Same Host

Send	Receive	Verify	Total
791	1803	138	2732
768	1537	216	2521
765	1626	229	2620
842	1569	163	2574
839	1679	245	2763
866	1672	241	2779
848	1743	163	2754
837	1625	240	2702
847	1660	241	2748
842	1714	162	2718
826	1661	206	2701

TABLE C.58: Java Sender to Java Receiver, 64 Megabyte Message, Different Hosts

Send	Receive	Verify	Total
844	7292	139	8275
801	7184	216	8201
779	7214	216	8209
804	7077	139	8020
799	7197	216	8212
803	7108	215	8126
800	7198	139	8137
794	7153	216	8163
799	7109	216	8124
795	7172	140	8107
799	7166	187	8159