mpiJava: An Object-Oriented Java interface to MPI

Mark Baker[1], Bryan Carpenter[2], Geoffrey Fox[2], Sung Hoon Ko[2] and Sang Lim[2]

[1] School of Computer Science
University of Portsmouth,
Southsea, Hants,
UK, PO4 8JF
Mark.Baker@port.ac.uk
[2] NPAC at Syracuse University
Syracuse, New York,
NY 13244, USA
{dbc,gcf,shko,slim}@npac.syr.edu

Abstract. A basic prerequisite for parallel programming is a good communication API. The recent interest in using Java for scientific and engineering application has led to several international efforts to produce a message passing interface to support parallel computation. In this paper we describe and then discuss the syntax, functionality and performance of one such interface, **mpiJava**, an object-oriented Java interface to MPI. We first discuss the design of the **mpiJava** API and the issues associated with its development. We then move on to briefly outline the steps necessary to 'port' **mpiJava** onto a range of operating systems, including Windows NT, Linux and Solaris. In the second part of the paper we present and then discuss some performance measurements made of communications bandwidth and latency to compare **mpiJava** on these systems. Finally, we summarise our experiences and then briefly mention work that we plan to undertake.

1 Introduction

It is generally recognised that the vast majority of scientific and engineering applications are written in either C, C++ or Fortran. The recent popularity of Java has led to it being seriously considered as a good language to develop scientific and engineering applications, and in particular for parallel computing[1][2][3][4]. Sun's claims, on behalf of Java, that it is simple, efficient and platform-neutral - a natural language for network programming - makes it attractive to scientific programmers who wish to harness the collective computational power of parallel platforms as well as networks of workstations or PCs, with interconnections ranging from LANs to the Internet. The attractiveness of Java for scientific computing is being encouraged by bodies like Java Grande[5]. The Java Grande forum has been set up to co-ordinate the communities efforts to standardise many aspects of Java and so ensure that its future development makes it more appropriate for scientific programmers.

Developers of parallel applications generally use the Single Program Multiple Data (SPMD) model of parallel computing, wherein a group of processes cooperate by executing identical program images on local data values. A programmer using the SPMD model has a choice of explicit or implicit means to move data between the cooperating processes. Today, the normal explicit means is via message passing and the implicit means is via data-parallel languages, such as HPF. Although using implicit means to develop parallel applications is generally thought to be easier, explicit message passing is more often used. The reasons for this are beyond the scope of this paper, but currently developers can produce more efficient and effective parallel applications using message passing.

A basic prerequisite for message passing is a good communication API. Java comes with various ready-made packages for communication, notably an interface to BSD sockets, and the Remote Method Invocation (RMI) mechanism. Both these communication models are optimised for client-server programming, whereas the parallel computing world is mainly concerned with 'symmetric' communication, occurring in groups of interacting peers.

This symmetric model of communication is captured in the successful Message Passing Interface standard (MPI), established a few years ago[6]. MPI directly supports SPMD model of parallel computing. Reliable point-to-point communication is provided through a shared, group-wide communicator, instead of socket pairs. MPI allows numerous blocking, non-blocking, buffered or synchronous communication modes. It also provides a library of true collective operations (broadcast is the most trivial example). An extended standard, MPI 2, allows for dynamic process creation and access to memory in remote processes.

The existing MPI standards specify language bindings for Fortran, C and C++. In this article we discuss a binding of MPI 1.1 for Java, and describe an implementation using Java wrappers to invoke C MPI calls through the Java Native Interface[8]. The software is publically available from:

`http://www.npac.syr.edu/projects/pcrc/mpiJava`

1.1 Related work

Early work by two of the current authors on Java MPI bindings is reported in[9][10]. In these papers we compared various approaches to parallel programming in Java, including sockets and MPI programming. A comparable approach to creating full Java

MPI interfaces is used by JCI[4], the *Java-to-C interface generator*. In JCI, Java wrappers are automatically generated from the C MPI header. This eases the implementation work, but does not lead to a fully object-oriented API.

MPIJ is a completely Java-based implementation of MPI which runs as part of the Distributed Object Group Metacomputing Architecture[12] (DOGMA). Being closely modelled on the MPI-2 C++ bindings, MPIJ implements a large subset of MPI functionality. MPIJ communication uses native marshalling of primitive Java types. This technique allows MPIJ to achieve communications speeds comparable with native MPI implementations. `jmpi`[13] is an MPI environment built upon JPVM[14], a Java-based implementation of PVM. `jmpi`, offers a full Java API to MPI 1.1 as well as features such as thread safety and multiple communication end-points per task. `jmpi` is a pure Java MPI environment, but is complicated by the need to call JPVM methods. Another recently announced Java MPI interface, called JavaWMPI[15], is built upon the Windows MPI environment WMPI[20]. JavaWMPI has a very similar in structure to `mpiJava`, but the syntax of the interface is less object-oriented and a procedural method is used to perform polymorphism between Java datatypes.

MPI Software Technology, Inc. has also announced their intention to deliver a commercial Java interface to MPI called JMPI[16]. Java implementations of the related PVM message-passing environment have been reported by Yalamanchilli et. al.[17] and The MPI Forum[6]. Many of the above-mentioned groups, as part of the Java Grande forum activities, have recently published a position paper[18] in an attempt to standardise on a single API.

1.2 Overview of this article

First we outline the `mpiJava` API and describe various special issues that arise in Java. Implications of object serialization are also explored briefly as are the difficulties due to the lack of true multidimensional arrays in Java.

This discussion is followed by a description of an implementation of the proposed Java binding through a set of wrappers that use the JNI to call existing MPI implementations. The virtues and problems of this implementation strategy are discussed, and results of tests and benchmarks on Solaris, Windows NT and Linux are presented.

2 Introduction to the `mpiJava` API

The MPI standard is explicitly object-based. The C and Fortran bindings rely on 'opaque objects' that can be manipulated only by acquiring object handles from constructor functions, and passing the handles to suitable functions in the library. The C++ binding specified in the MPI 2 standard collects these objects into suitable class hierarchies and defines most of the library functions as class member functions. The `mpiJava` API follows this model, lifting the structure of its class hierarchy directly from the C++ binding. The major classes of `mpiJava` are illustrated in Figure 1.

The class MPI only has static members. It acts as a module containing global services, such as initialization of MPI, and many global constants including the default communicator `COMM_WORLD`.

The most important class in the package is the communicator class `Comm`. All communication functions in `mpiJava` are members of `Comm` or its subclasses. As usual in MPI, a communicator stands for a 'collective object' logically shared

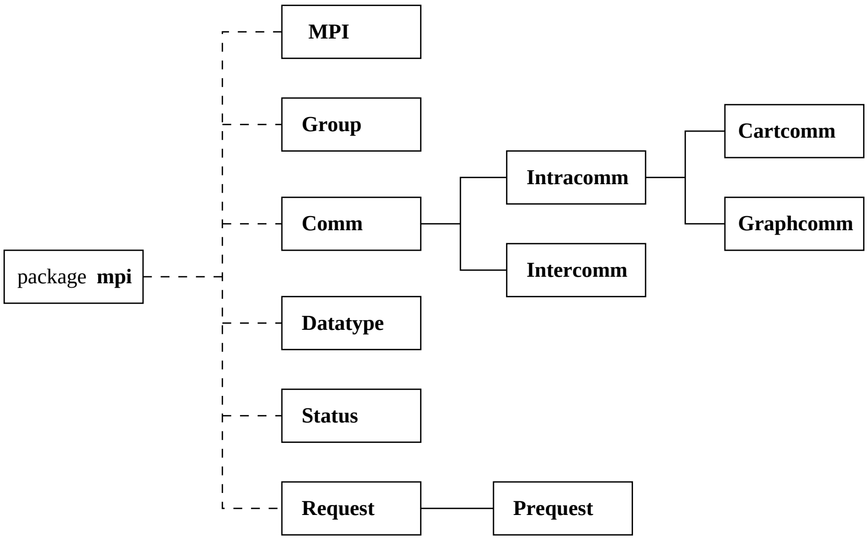

Fig. 1. Principal classes of `mpiJava`

by a group of processors. The processes communicate, typically by addressing messages to their peers through the common communicator.

Another class that is important for the discussion below is the `Datatype` class. This describes the type of the elements in the message buffers passed to send, receive, and all other communication functions. Various basic datatypes are predefined in the package. These mainly correspond to the primitive types of Java, shown in Figure 2.

MPI datatype	Java datatype
MPI.BYTE	`byte`
MPI.CHAR	`char`
MPI.SHORT	`short`
MPI.BOOLEAN	`boolean`
MPI.INT	`int`
MPI.LONG	`long`
MPI.FLOAT	`float`
MPI.DOUBLE	`double`
MPI.PACKED	

Fig. 2. Basic datatypes of `mpiJava`

The standard send and receive operations of MPI are members of `Comm` with interfaces:

```
public void Send(Object buf, int offset, int count, Datatype
                 datatype, int dest, int tag)

public Status Recv(Object buf, int offset, int count, Datatype
                   datatype, int source, int tag)
```

In both cases the actual argument corresponding to buf must be a Java array. In the current implementation they must be arrays with elements of primitive type. By implication they must be one-dimensional arrays, because Java 'multidimensional arrays' are really arrays of arrays. In these and all other `mpiJava` calls, the buffer array argument is followed by an offset that specifies the element of in array where the message actually starts.

2.1 Special features of the Java binding

The `mpiJava` API is modelled as closely as practical on the C++ binding defined in the MPI 2.0 standard (currently we only support the MPI 1.1 subset). A number of changes to argument lists are forced by of the restriction that arguments cannot be passed by reference in Java. In general outputs of `mpiJava` methods come through the result value of the function. In many cases MPI functions return more than one value. This is dealt with in `mpiJava` in various ways. Sometimes an MPI function initializes some elements in an array and also returns a count of the number of elements modified. In Java we typically return an array result, omitting the count. The count can be obtained subsequently from the `length` member of the array. Sometimes an MPI function initializes an object conditionally and returns a separate flag to say if the operation succeeded. In Java we return an object handle which is `null` if the operation fails. Occasionally an extra field is added to an existing MPI class to hold extra results - for example the `Status` class has an extra field, index, initialized by functions like `Waitany`. Rarely none of these methods work and we resort to defining auxilliary classes to hold multiple results from a particular function. In another change to C++, we often omit array size arguments, because they can be picked up within the wrapper by reading the `length` member of the array argument.

As a result of these changes `mpiJava` argument lists are often more concise than the corresponding C or C++ argument lists.

Normally in `mpiJava`, MPI destructors are called by the Java `finalize` method for the class. This is invoked automatically by the Java garbage collector. For most classes, therefore, no binding of the `MPI_class_FREE` function appears in the Java API. Exceptions are `Comm` and `Request`, which do have explicit `Free` members. In those cases the MPI operation could have observable side-effects (beyond simply freeing resources), so their execution is left under direct control of the programmer.

```java
// Simple program

import mpi.*;

class Hello {
  static public void main(String[] args){
    MPI.Init(args);
    int myrank = MPI.COMM_WORLD.Rank();
    if(myrank == 0)}
      char [] message = "Hello, there".toCharArray() ;
      MPI.COMM_WORLD.Send(message,0,message.length, MPI.CHAR, 1, 99);
    } else {
      char [] message = new char [20] ;
      MPI.COMM_WORLD.Recv(message, 0, 20, MPI.CHAR, 0, 99) ;
      System.out.println("received:" + new String(message) + ":");

    }
    MPI.Finalize();
  }
}
```

Fig. 3. Minimal `mpiJava` program (run in two processes)

2.2 Derived datatype vs Object serialization

In MPI new *derived types* of class `Datatype` can be created using suitable library functions. The derived types allow one to treat contiguous, strided, or indirectly indexed segments of program arrays as individual message elements. The corresponding array subsections can then be communicated in a single function call, potentially exploiting any special hardware or software the platform provides for exchanging scattered data between user space and the communication system.

Currently `mpiJava` provides all the derived datatype constructors of standard MPI, with one limitation: it places significant restrictions on its binding of `MPI_TYPE_STRUCT`. In C or Fortran this function can be used to describe an entity combining fields of different primitive (or derived) type. Because of the assumption that buffers are one-dimensional arrays with elements of primitive type, `mpiJava` imposes a restriction that all the types combined by its `Datatype.Struct` member must have the same *base type*, which must agree with the element type of the buffer array. Also `mpiJava` does not provide an analogue of `MPI_BOTTOM` buffer address, or the `MPI_ADDRESS` function for finding offsets relative to this absolute member base. In C or Fortran these functions allow buffers to include fields from separately declared variables or arrays, but the mechanism does not sit well with the pointer-free Java language model.

Approaches based on the MPI derived datatype model do not seem to be the best way to alleviate this restriction. A better option is probably to exploit the run-time type information already provided in Java objects. We are developing a version of `mpiJava` that adds one new predefined datatype:

`MPI.Object`

A message buffer can then be an array of any serializable Java objects. The objects are serialized automatically in the wrapper of send operations, and unserialized at their destination.

The absence of true multi-dimensional arrays in Java limits another use of derived data types. In MPI the `MPI_TYPE_VECTOR` function creates a derived datatype representing a strided section of an array. In C or Fortran this strided section can be identified with a section of a multi-dimensional array. It could describe, say, an edge of the local patch of a two-dimensional distributed array. In Java there is no equivalence between a multi-dimensional array and a contiguous patch of memory, or a one-dimensional array. The programmer may choose to linearize all multi-dimensional arrays in the algorithm, representing them as one-dimensional arrays with suitable index expressions. In this case derived datatypes can be used to send and receive sections of the array. Alternatively the programmer may use Java arrays of arrays to represent multi-dimensional arrays. This simplifies the index arithmetic in the program. Sections of the array are then explicitly copied to one-dimensional buffers for communication. The latter option seems to be more popular with programmers.

Although, for reasons of conformance of with MPI standards, we expect to continue supporting derived datatypes in `mpiJava`, their value in the Java domain is less clear-cut than in C or Fortran. Allowing serializable objects as buffer elements is probably a more powerful facility.

3 `mpiJava` implementations on PC Platforms

3.1 Introduction

As Java is a platform-neutral language there is much interest in 'porting' `mpiJava` to PC-based systems, in particular Windows NT and Linux. To 'port' `mpiJava` onto a PC environment it is necessary to have a native MPI library, a version of the Java Development Toolkit (JDK) and a C compiler. `mpiJava` consists of two main parts: the MPI Java classes and the C stubs that binds the MPI Java classes to the underlying native MPI implementation. We create these C stubs using JNI - the means by which Java can call and pass parameters to and from a native API. Figure 4 provides a simple schematic view of the software layers involved.

To 'port' `mpiJava` onto a new platform, generally two steps need to be undertaken:

- Create a native library out of the compiled JNI C stubs.
- Compile MPI Java in class libraries - ensuring that the correctly named stub library is loaded by the Java `System.loadLibrary("stublib")` call in the main source file `MPI.java`.

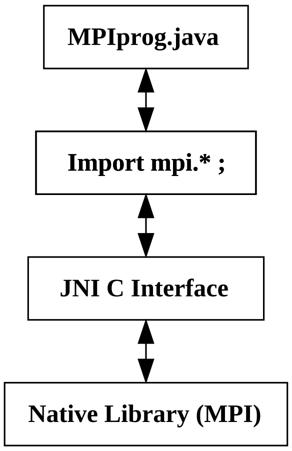

Fig. 4. Software Layers

The development and testing of `mpiJava` was undertaken on various Sun and SGI UNIX platforms using MPICH. As interfacing Java to MPI is not always trivial, in earlier implementation we often saw low-level conflicts between the Java runtime and the interrupt mechanisms used in the MPI implementations. The situation is improving as the JDK matures, in particular version 1.2β allows the use of *green* or native threads, which have eliminated the interrupt problem that we encountered with earlier releases of the JDK. `mpiJava` is now stable on NT platforms using WMPI and JDK 1.1 or later as well as UNIX platforms using MPICH and JDK 1.2β.

3.2 Windows NT

To test `mpiJava` under Windows NT we had the choice of a number of MPI implementations to pick from[19]. We chose WMPI from the Instituto Supererior de Engenharia de Coimbra, Portugal. WMPI is a full implementation of MPI for Microsoft Win32 platforms. WMPI is based on MPICH and includes a p4[21] device standard. P4 provides the communication internals and a startup mechanism. The WMPI package is a set of libraries (for Borland C++, Microsoft Visual C++ and Microsoft Visual FORTRAN). The release of WMPI provides libraries, header files, examples and daemons for remote start-up. WMPI can co-exist and interact with MPICH/ch_p4 in a cluster of mixed UNIX and Win32 platforms. WMPI is still under development and is freely available.

mpiJava under WMPI: To create a release of `mpiJava` for WMPI the following steps were undertaken:

- *Step 1* - Compile the `mpiJava` JNI C interface into a Win32 Dynamic Link Library (`mpiJava.dll`).

– *Step 2* - Modify the name of the library loaded by the `mpiJava` interface (`MPI.java`) to that of the of the newly compiled library.
– *Step 3* - Compile the Java MPI interface into class libaries.
– *Step 4* - Create a JNI interface to WMPI. This was necessary as under WMPI a master process is first spawned. Its purpose is to first read in a job configuration file and use the information within it to set up and run the actual MPI processes. An idiosyncrasy of WMPI is that all MPI processes must have a file name with the extension *.EXE*. This led to the need to produce a JNI to WMPI so that the JVM was loaded and the 'main' method of `mpiJava` Java class started.

3.3 Linux

At the time of writing this paper, our attempts to 'port' `mpiJava` to Linux are in progress. We are currently experiencing problems similar to those encountered during our early attempts to create the interface on Solaris, mentioned in section 3.1 . Sun's releases the JDK for Solaris and NT platforms first. On other platforms, such as Linux, it is necessary for developers to 'port' the JDK. The most recent release of the JDK for Linux is 1.1.7 and this is version is known to be the cause of our problems. It is anticipated that JDK 1.2 will be available for Linux shortly and that we will be able to report our experiences with Linux at the IPPS/SPDP 99 workshop in April 1999.

3.4 Functionality Tests

An integral part of the development of this project was to produce or translate a number of basic MPI test codes to `mpiJava`. An obvious starting point was the C test suite originally developed by IBM . This suite had been modified to comply fully with the MPI standard and to be compatible with the MPICH. The suite consists of fifty-seven C programs that test the following MPI calls and data types; collective operations, communicators, data types, environmental inquiries, groups, point to point and virtual topologies. These codes were all translated to `mpiJava`.

Under WMPI and Solaris-MPICH these codes were run either as multiple processes on a single machine (Shared Memory mode - SM) or as multiple processes running on separate machines (Distributed Memory mode - DM). Under WMPI and Solaris-MPICH all the codes ran in both modes without alterations. Our experiences using Linux-MPICH will be reported when JDK 1.2 is available for Linux.

4 Simple Communications Performance Measurement

4.1 Introduction

At this early stage of our project we have decided to restrict performance measurements to those that will give some indication of the basic inter-processor communications performance. The actual computational performance of each process is felt to be dependent on the local JVM and associated technologies used by specific vendors to increase the performance of Java.

4.2 PingPong Communications Performance Tests

In this program increasing sized messages are sent back and forth between processes - this is commonly called PingPong. This benchmark is based on standard blocking `MPI_Send/MPI_Recv`. PingPong provides information about latency of `MPI_Send/MPI_Recv` and uni-directional bandwidth. To ensure that anomalies in message timings are minimised the PingPong is repeated many times for each message size. The codes used for these tests were those developed by Baker and Grassl[23]. The three existing codes (MPI-C, MPI-Fortran and Winsock-C) were used for comparison and we implemented an `mpiJava` version for our purposes.

The main problem encountered running the PingPong code was that under WMPI on Win32 `MPI_Wtime()` had been implemented with a millisecond resolution. It was necessary to adapt each of the codes to use an alternative timer with microsecond (μs) resolution. The performance tests shown in the next section were run on two similar systems:

- Two dual processor (P6 200 MHz) NT 4 workstations each with 128 MBytes of DRAM.
- Two dual processor (UltraSparc 200 MHz) Solaris workstations with 256 MBytes of DRAM.

Both systems were connected via 10BaseT Ethernet and the tests were carried out when there was little network activity and on quiet machines.

4.3 Message Startup Latencies

	Wsock	**WMPI-C**	**WMPI-J**	**MPICH-C**	**MPICH-J**	**Linux-C**	**Linux-J**
SM	144.8 μs	67.2 μs	161.4 μs	148.7 μs	374.6 μs	- μs	- μs
DM	244.9 μs	623.9 μs	689.7 μs	679.1 μs	961.2 μs	- μs	- μs

Table 1. Time for 1 Byte Messages

In Table 1 we show the transmission time in microseconds (s) to send a 1 byte message in each of the environments tested. In SM the `mpiJava` wrapper adds an extra 94μs (140%) and 226μs (152%) compared to WMPI and MPICH C respectively. In DM the `mpiJava` wrapper adds an extra 66μs (11%) and 282μs (42%) compared to WMPI and MPICH C respectively. The Wsock figures are those for a WinSock implementation of PingPong benchmark using TCP. Linux results will be presented during the conference workshop.

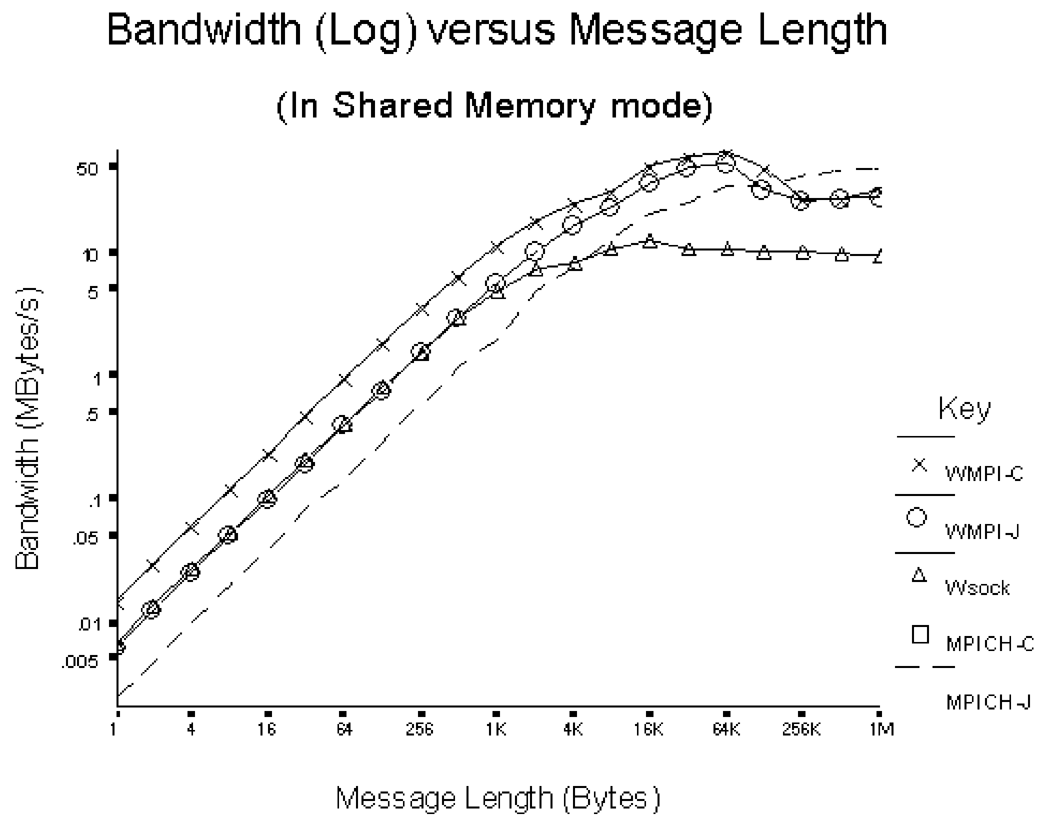

Fig. 5. PingPong Results in Shared Memory (SM) mode

4.4 Results in Shared Memory Mode (Figure 5)

The `mpiJava` curve mirrors that of C with an almost constant offset up to 8K, thereafter the curves converge meeting at 256K. Under MPICH, the curves for C and `mpiJava` mirror each other in a similar fashion to those under WMPI, again there is a constant offset and convergence at around 256K.

Under WMPI the peak bandwidth of C is around 65 MBytes/s and `mpiJava` is 54 MBytes/s. The peaks occur at around 64K. Under MPICH the bandwidth is flattening out, but still increasing for C and `mpiJava`, at the 1M. The actual rate measured at this point is about 50 MBytes/s.

Clearly the WMPI C code perform best of those tested. The performance of `mpiJava` in SM under WMPI is good - it exhibits a fairly constant overhead of $95\mu s$ up to 2K, thereafter it converges with the C curve. The performance the C code under MPICH is slightly surprising as the NT and Solaris platforms used for these tests had similar specifications. It is assumed that the performance reflects the usage of MPICH rather than a native version of MPI for Solaris. Even so, the MPICH results for `mpiJava` show that it exhibits reasonable performance.

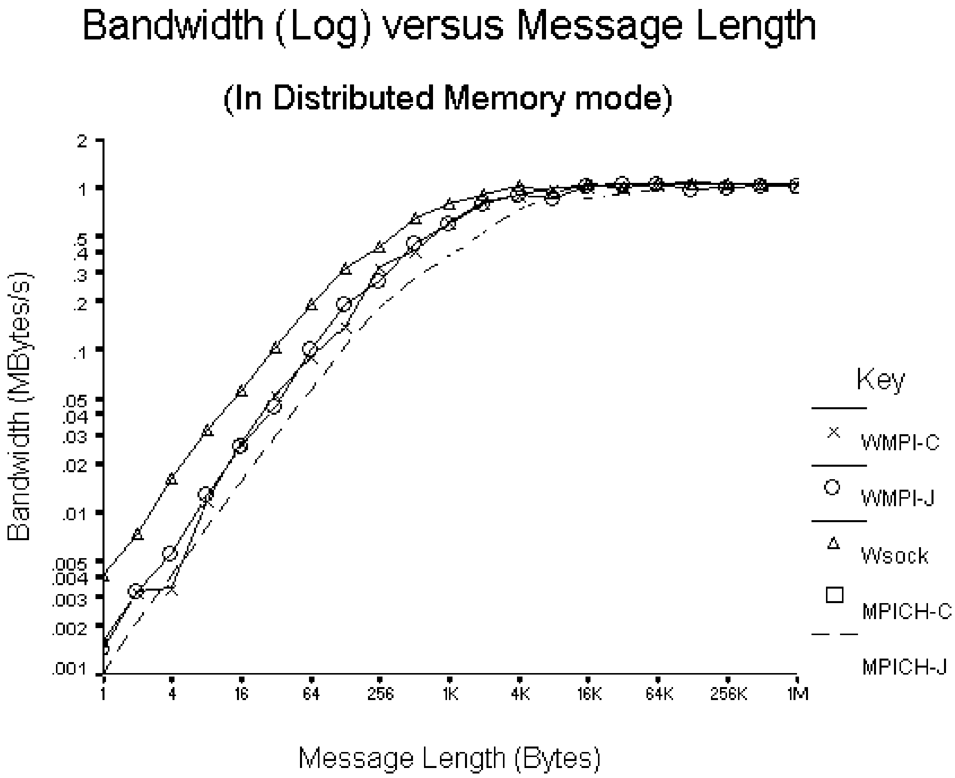

Fig. 6. PingPong Results in Distributed Memory (DM) mode

4.5 Results in Distributed Memory Mode (Figure 6)

In DM the differences between the MPI codes is not as pronounced as seen in SM. Under WMPI the C and `mpiJava` codes display very similar performance characteristics throughout the range tested. Under MPICH, there is distinct performance difference between C and `mpiJava`, However the difference is much smaller than in SM and the curves converge at the 4K. All curves peak at about 1 MByte/s, which is about 90% of the maximum attainable on 10 Mbps Ethernet link.

4.6 Overall Results Discussion

In both SM and DM modes `mpiJava` adds a fairly constant overhead compared to normal native MPI. In an environment like WMPI, which has been optimised for NT, the actual overheads of using `mpiJava` are relatively small at around 100ms. Under MPICH the situation is not quite so good, here the use of `mpiJava` introduces an extra overheads of between 250 - 300μs.

It should be noted that these results compare codes running directly under the operating system with those running in the JVM. For example, according to a single 200 MHz PentiumPro will achieve in excess of 62 Mflop/s on a Fortran version of LinPack. A test of the Java LinPack code gave a peak performance of 22 Mflop/s for the same processor running the JVM. The difference in performance will account for much of the additional overhead that `mpiJava` imposes on C MPI codes. From this it can be deduced that the quality and performance of JVM on each platform will have the greatest effect on the usefulness of `mpiJava` for scientific computation.

5 Conclusions

5.1 Overall Summary

We have discussed the design and development of `mpiJava` - a pure Java interface to MPI. We have also highlighted the benefits of a fully object-oriented Java API compared to those currently available. Our performance tests have shown that `mpiJava` should fulfil the needs of MPI programmers not only in terms of functionality but also in terms of good performance when compared to similar C MPI programs. Unfortunately, at the time of submission of this paper we have been unable to test `mpiJava` under Linux, but we believe that the problems we have encountered we be overcome soon and we will be able to present our finding during the Workshop in April 1999. Overall, however, we feel that we have implemented a well designed, functional and efficient Java interface to MPI.

5.2 Particular Conclusions

– `mpiJava` provides a fully functional and efficient Java interface to MPI.
– Our performance tests have shown that, in terms of communications speeds, WMPI on NT out performs MPICH on Solaris.
– When used for distributed computing the current implementation of `mpiJava` does not impose a huge overhead on-top of the native MPI interface.
– We have discovered some of the limitation in the usage of JNI. In particular with MPICH where we had problems with UNIX signals. We are hopeful that these problems will disappear when we start using JDK 1.2 and native threads.
– Our performance tests indicate that much of the additional latency that `mpiJava` imposes is due to the relatively poor performance of the JVM rather than the impact of messages traversing additional software layers.
– The syntax of `mpiJava` is easy to understand and use, thus making it relatively simple for programmers with either a Java or Scientific background to take up.
– We believe that `mpiJava` will also provide a popular means for teaching students the fundamentals of parallel programming with MPI.

5.3 Future Work

We plan to continue to improve `mpiJava` with further Java features, such as object serialization, and also add-in the functionality that has been proposed in MPI-2. We intend to 'port' `mpiJava` to a multitude of new MPI environments, including LAM, Sun MPI and Globus. We are also planning a pure-Java MPI environment which does not rely on native MPI services.

References

1. *Parallel Compiler Runtime Consortium, HPCC and Java* - a report by the Parallel Compiler Runtime Consortium, `http://www.npac.syr.edu/users/gcf/hpjava3.html`, May 1996.
2. G.C. Fox, editor, *Java for Computational Science and Engineering - Simulation and Modelling II*, volume 9(11) of Concurrency: Practice and Experience, November 1997.
3. G.C. Fox, editor, *ACM 1998 Workshop on Java for High-Performance Network Computing*, Palo Alto, February 1998, Concurrency: Practice and Experience, 1998.
4. V. Getov, S. Flynn-Hummel, and S. Mintchev. *High-Performance parallel programming in Java: Exploiting native libraries*, In ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998, Concurrency: Practice and Experience, 1998. To appear.
5. Java Grande Forum - `http://www.javagrande.org/`
6. *Message Passing Interface Forum. MPI: A Message-Passing Interface Standard*, University of Tennessee, Knoxville, TN, June 1995. `http://www.mcs.anl.gov/mpi`
7. G.C. Fox, editor, *Java for Computational Science and Engineering - Simulation and Modelling*, volume 9(6) of Concurrency: Practice and Experience, June 1997.
8. R. Gordon, *Essential JNI: Java Native Interface*, Prentice Hall, 1998.
9. D.B. Carpenter, Y. Chang, G.C. Fox, D. Leskiw, and X. Li, *Experiments with HP-Java* Concurrency: Practice and Experience, 9(6):633, 1997.
10. D.B. Carpenter, Y. Chang, G.C. Fox, and X. Li, *Java as a language for scientific parallel programming*, In 10th International Workshop on Languages and Compilers for Parallel Computing, volume 1366 of LNCS, pages 340–354, 1997.
11. S. Mintchev and V. Getov, *Towards portable message passing in Java: Binding MPI, Recent Advances in MPI and PVM*, Editors, M. Bubak, J. Dongarra and J. Wasniewski, volume 1332 of LNCS pages 135 - 142, Springer Verlag, 1997.
12. DOGMA - `http://zodiac.cs.byu.edu/DOGMA/`
13. K. Dincer and K. Ozbas, *jmpi and a Performance Instrumentation Analysis and Visualization Tool for jmpi*, 1st UK Workshop on Java for High Performance Network Computing, Southampton, UK, September 1998.
14. A.J. Ferrari, *JPVM: Network parallel computing in Java*, In ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998, Concurrency: Practice and Experience, 1998. To appear.
15. P. Martin, L.M. Silva and J.G. Silva, *A Java Interface to MPI*, Proceeding of the 5th European PVM/MPI Users' Group Meeting, Liverpool UK, September 1998 WMPI - `http://dsg.dei.uc.pt/w32mpi/`
16. G. Crawford III, Y. Dandass, and A. Skjellum, *The JMPI commercial message passing environment and specification: Requirements, design, motivations, strategies, and target users*, `http://www.mpi-softtech.com/publications`

17. N. Yalamanchilli and W. Cohen, *Communication performance of Java based parallel virtual machines*, In ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998, Concurrency: Practice and Experience, 1998. To appear.
18. B. Carpenter, V. Getov, G. Judd, T. Skjellum and G. Fox, *MPI for Java - Position Document and Draft API Specification*, November 1998 - `http://www.npac.syr.edu/projects/pcrc/mpiJava`
19. M.A. Baker and G.C. Fox, *MPI on NT: A Preliminary Evaluation of the Available Environments*, 12th IPPS & 9th SPDP, LNCS, Jose Rolim (Ed.), Parallel and Distributed Computing, Springer Verlag, Heidelberg, Germany. ISBN 3-540 64359-1, April 1998.
20. WMPI - `http://dsg.dei.uc.pt/w32mpi/`
21. R. Butler and E. Lusk, *Monitors, messages, and clusters: The p4 parallel programming system*, Parallel Computing, 20:547–564, April 1994.
22. IBM Test Suite - `ftp://info.mcs.anl.gov/pub/mpi/mpi-test/ibmtsuite.tar`
23. PingPong Benchmarks - `http://www.dcs.port.ac.uk/~mab/TOPIC/`
24. LinPack - `http://performance.netlib.org/performance/html/linpack.data.col0.html`
25. Java LinPack - `http://www.netlib.org/benchmark/linpackjava/`