mpiJava: A Java Interface to MPI

Mark Baker
University of Portsmouth
Southsea
Hants, UK, PO4 8JF
mab@sis.port.ac.uk

Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, Xinying Li
NPAC at Syracuse University,
Syracuse,
NY 13244, USA
{dbc,gcf,shko,xli}@npac.syr.edu

February 8, 2000

Abstract

The overall aim of this paper is to introduce *mpiJava*—a Java interface to the widely used Message Passing Interface (MPI). In the first part of the paper we discuss the design of the mpiJava API and issues associated with its development. In the second part of the paper we briefly describe an implementation of mpiJava on NT using the WMPI environment. We then discuss some measurements made of communications performance to compare mpiJava with C and Fortran bindings of MPI. In the final part of the paper we summarize our findings and briefly mention work we plan to undertake in the near future.

1 Introduction

Recently there has been a great deal of interest in the idea that Java may be a good language for scientific and engineering computation, and in particular for parallel computing [5, 10, 11, 12]. The claims made on behalf of Java, that it is simple, efficient and platform-neutral—a natural language for network programming—make it potentially attractive to scientific programmers hoping to harness the collective computational power of networks of workstations and PCs, or even of the Internet.

A basic prerequisite for parallel programming is a good communication API. Java comes with various ready-made packages for communication, notably an easy-to-use interface to BSD sockets, and the Remote Method Invocation (RMI) mechanism. Interesting as these interfaces are, it is questionable whether parallel programmers will find them especially convenient. Sockets and remote procedure calls have been around for about as long as parallel computing has been fashionable, and neither of them has been popular in that field. Both communication models are optimized 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 [9]. MPI directly supports 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. 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 [14]. 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 [3, 4]. In those papers we compared various approaches to parallel programming in Java, including socket programming and MPI programming. A comparable approach to creating full Java MPI interfaces has been taken by Getov and Mintchev [17, 13]. In their work Java wrappers were automatically generated from the C MPI header. This eases the implementation work, but does not lead to a fully object-oriented API. A subset of MPI is implemented in the DOGMA system for Java-based parallel programming [16]. MPI Software Technology, Inc have announced their intention to deliver a commercial Java interface to MPI called JMPI [15]. Java implementations of the related PVM message-passing environment have been reported in [18] and [8].

1.2 Overview of this article.

First we outline the the API and describe various special issues that arise in Java. Implications of object serialization for the Java MPI interface are explored

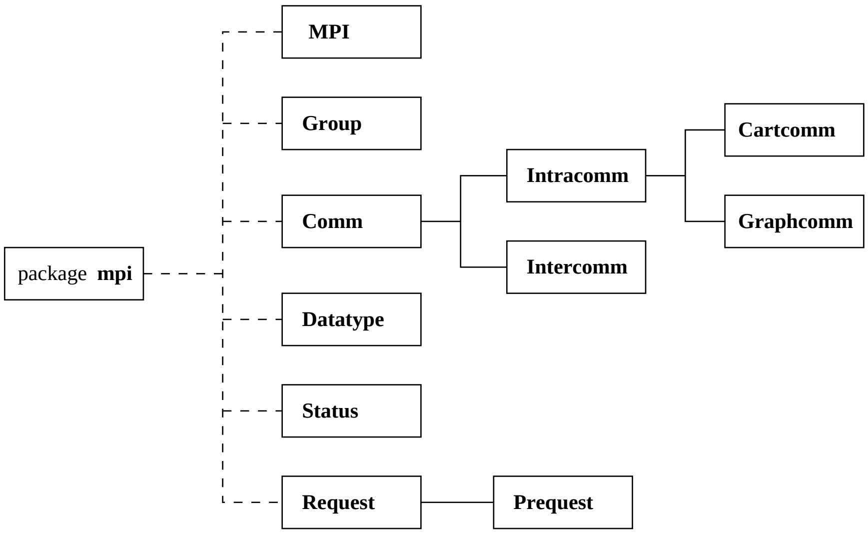

Figure 1: Principal classes of mpiJava

briefly. Difficulties due to the lack of true multidimensional arrays in Java are mentioned.

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

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	

Figure 2: Basic datatypes of mpiJava

usual in MPI, a communicator stands for a "collective object" logically shared 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.

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 "multi-dimensional 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 in the 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

```
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();
  }
}
```

Figure 3: Minimal mpiJava program (run in two processes)

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.

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 memory base. In C or Fortran these functions allow buffers to include fields from separately declared variables or arrays, but the mechanism does not sit very 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

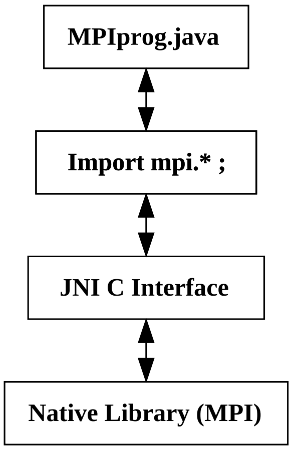

Figure 4: Software Layers

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 clearcut than in C or Fortran. Allowing serializable objects as buffer elements is probably a more powerful facility.

3 mpiJava implementation on WMPI

3.1 Introduction

A crucial part of mpiJava is the JNI C interface library. This library has the JNI stubs that bind the Java MPI calls to the underlying native MPI library—Figure 4 provides a simple schematic view of the software layers involved.

The development and testing of mpiJava was undertaken on various Sun and SGI UNIX platforms using MPICH. A number of problems were encountered with MPI on these systems which was generally caused by the interaction of the mpiJava JNI and the native MPI library. These problems led us to explore a number of different MPI environments, including those for NT. Currently, WMPI [21] on NT is the only environment that enables mpiJava to work consistently and efficiently. In this section of the paper we will describe the WMPI implementation of mpiJava for NT.

3.2 WMPI

WMPI from the Instituto Supererior de Engenharia de Coimbra, Portugal is a full implementation of MPI for Microsoft Win32 platforms. WMPI is based on MPICH and includes a P4 [2] device standard. P4 provides 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 freely available. WMPI and other MPI environments for NT had been explored thoroughly in a previous project [1].

3.3 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 Java interface (`MPI.java`) to that of the newly compiled library.

Step 3: 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 interface to WMPI so that the JVM was loaded and the "main" method of mpiJava Java class started. Once these three steps had been completed mpiJava for WMPI was ready for testing.

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 [19]. This suite had been modified to comply fully with the MPI standard and to be compatible with MPICH. The suite consists of fifty-seven C programs that test the following MPI calls and data types: collective operations, communicators, data types, enviromental inquiries, groups, point to point and virtual topologies. These codes were all translated to mpiJava.

Under WMPI these codes were run either as multiple processes on a single machine (Shared Memory mode—SM) or as multiple processes running on separate machines (Distribute Memory mode—DM). Under WMPI all the codes ran in both modes without alterations being made apart from specifying that more global memory was needed by WMPI.

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 minimized the PingPong is repeated many times for each message size. The codes used for these tests were those developed by Baker and Grassl [20]. 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 times 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.

	WMPI-C	WMPI-J	MPICH-C	MPICH-J
SM	67.2 μs	161.4 μs	148.7 μs	374.6 μs
DM	623.3 μs	689.7 μs	679.1 μs	961.2 μs

Table 1: Times for 1byte messages.

4.3 Message Start up latencies

In Table 1 we show the transmission time in microseconds to send a 1 byte message in each of the environments tested. In SM the mpiJava wrapper adds an extra 94 μs (140%) and 226 μs (152%) compared with WMPI and MPICH C respectively. In DM the mpiJava wrapper adds and extra 66 μs (11%) and 282 μs (42%) compared to WMPI and MPICH C respectively.

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 μs 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 the native version of MPI for Solaris. Even so, the MPICH results for mpiJava show that it exhibits reasonable performance.

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 10Mbps 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 optimized for NT, the actual overheads of using mpiJava are relatively small at around 100μs. 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 systems with those running in the JVM. For example, according to [6] 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 [7] 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. Overall, 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 communication speeds, WMPI on NT outperforms 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 limitations in the usage fo 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 of 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] Mark Baker and Geoffrey Fox. MPI on NT: A preliminary evaluation of available environments. In *12th International Parallel Processing Symposium & 9th Symposium on Parallel and Distributed Processing*, Lecture Notes in Computer Science, Heidelberg, Germany, April 1998. Springer Verlag.

[2] Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20:547–564, April 1994.

[3] Bryan Carpenter, Yuh-Jye Chang, Geoffrey Fox, Donald Leskiw, and Xiaoming Li. Experiments with HPJava. *Concurrency: Practice and Experience*, 9(6):633, 1997.

[4] Bryan Carpenter, Yuh-Jye Chang, Geoffrey Fox, and Xiaoming Li. Java as a language for scientific parallel programming. In *10th International Workshop on Languages and Compilers for Parallel Computing*, volume 1366 of *Lecture Notes in Computer Science*, pages 340–354, 1997.

[5] 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.

[6] Jack Dongarra. Linpack benchmark. http://performance.netlib.org/performance/html/linpack.data.col0.html.

[7] Jack Dongarra and Reed Wade. Linpack benchmark—Java version. http://www.netlib.org/benchmark/linpackjava/.

[8] Adam 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.

[9] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.* University of Tenessee, Knoxville, TN, June 1995. http://www.mcs.anl.gov/mpi.

[10] Geoffrey C. Fox, editor. *Java for Computational Science and Engineering—Simulation and Modelling*, volume 9(6) of *Concurrency: Practice and Experience*, June 1997.

[11] Geoffrey C. Fox, editor. *Java for Computational Science and Engineering—Simulation and Modelling II*, volume 9(11) of *Concurrency: Practice and Experience*, November 1997.

[12] Geoffrey C. Fox, editor. *ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998*, Concurrency: Practice and Experience, 1998. To appear. http://www.cs.ucsb.edu/conferences/java98.

[13] Vladimir Getov, Susan Flynn-Hummel, and Sava 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.

[14] Rob Gordon. *Essential JNI: Java Native Interface.* Prentice Hall, 1998.

[15] George Crawford III, Yoginder Dandass, and Anthony Skjellum. The JMPI commercial message passing environment and specification: Requirements, design, motivations, strategies, and target users. http://www.mpi-softtech.com/publications.

[16] Glenn Judd, Mark Clement, and Quinn Snell. DOGMA: Distributed object group management architecture. In *ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998*, Concurrency: Practice and Experience, 1998. To appear.

[17] Sava Mintchev and Vladimir Getov. Towards portable message passing in Java: Binding MPI. Technical Report TR-CSPE-07, University of Westminster, School of Computer Science, Harrow Campus, July 1997.

[18] Narendar Yalamanchilli and William 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.

[19] IBM test suite. ftp://info.mcs.anl.gov/pub/mpi/mpi-test/ibmtsuite.tar.

[20] PingPong benchmarks. http://www.dcs.port.ac.uk/ mab/TOPIC/.

[21] WMPI. http://dsg.dei.uc.pt/w32mpi/.

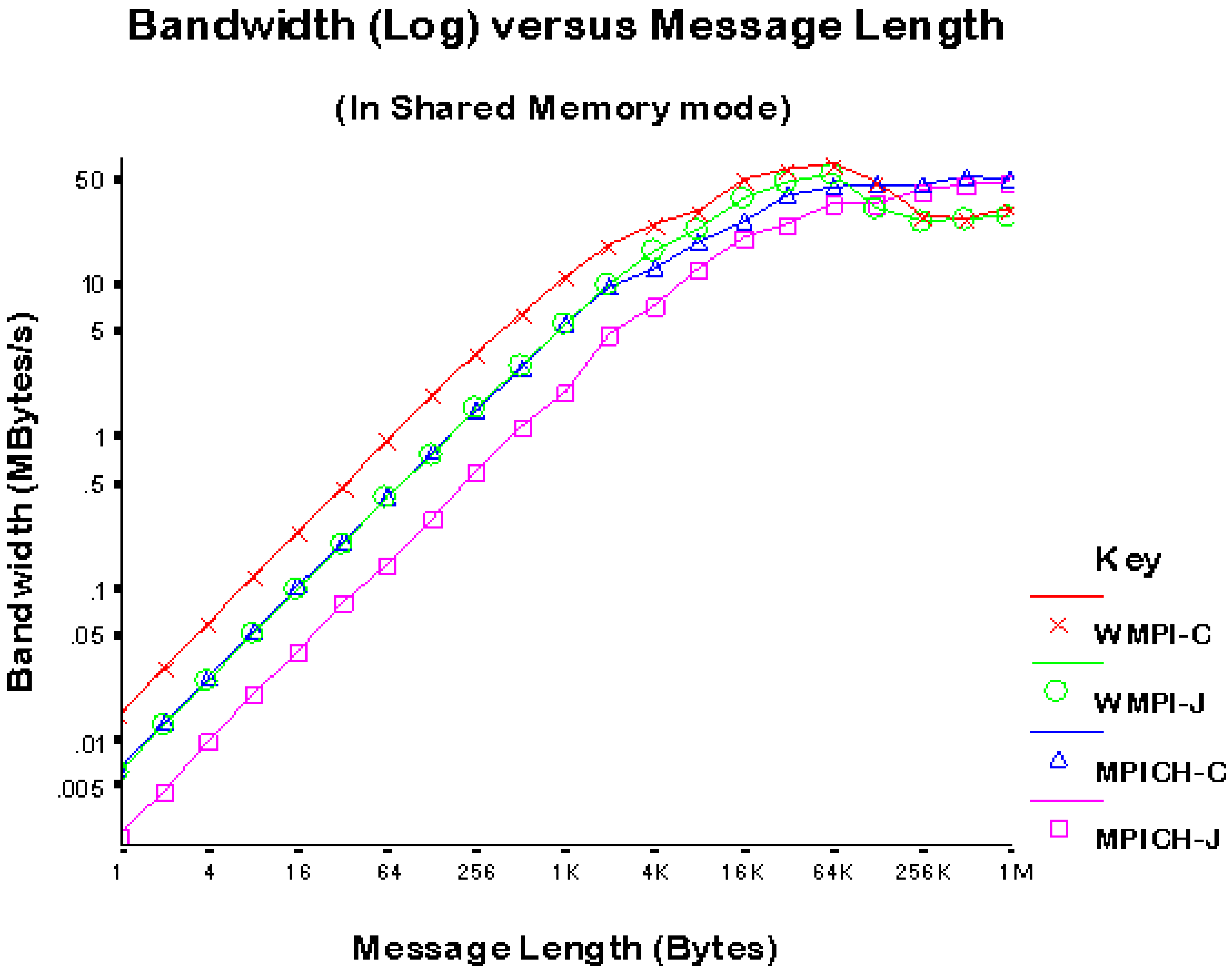

Figure 5: PingPong Results in Shared Memory mode

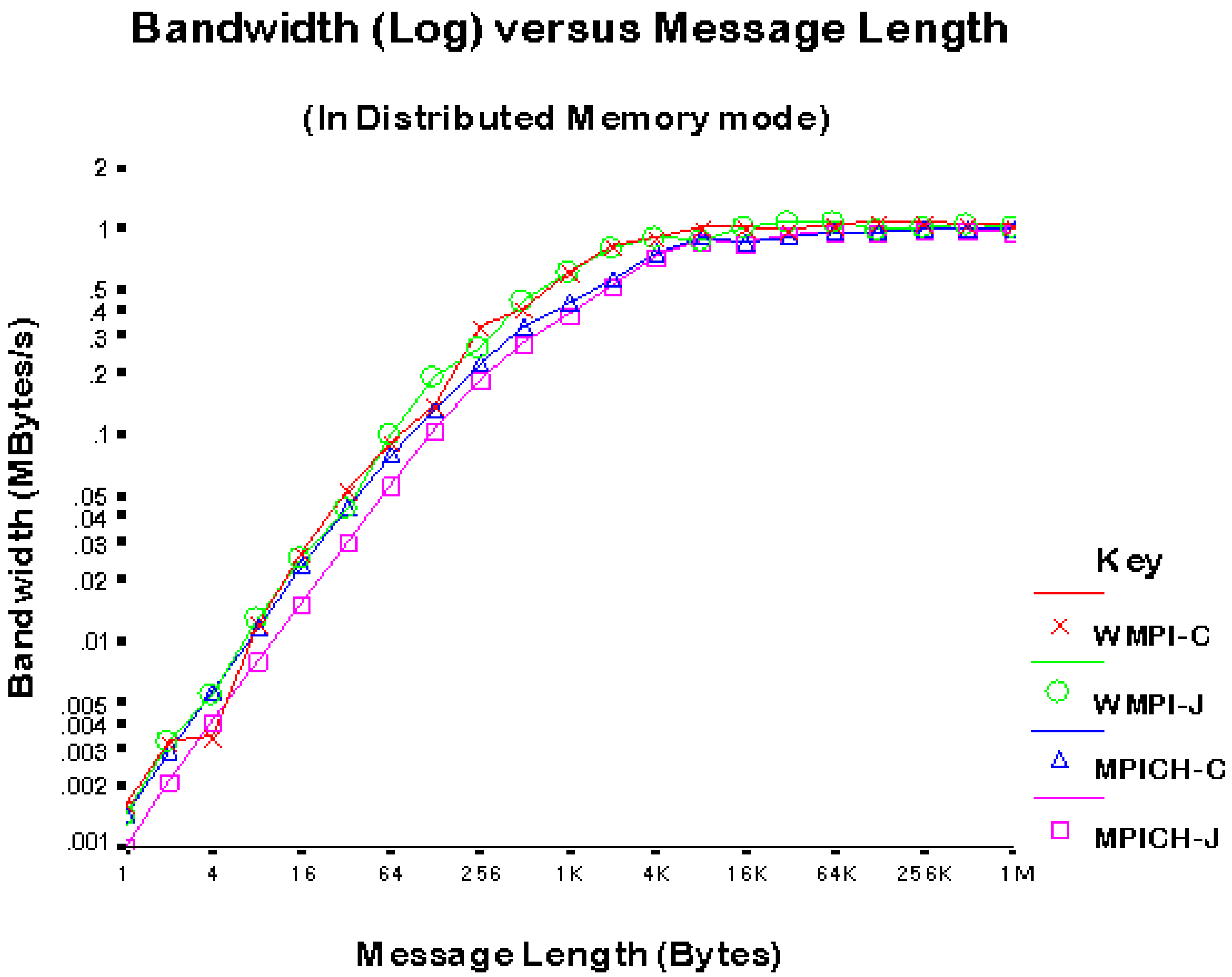

Figure 6: PingPong Results in Distributed Memory mode