

jmp: A Pure Java Implementation of MPI

Kivanc Dincer, Emrah Billur, Kadriye Ozbas

*Department of Computer Engineering
Başkent University
Bağlıca Campus, Eskisehir Road 20. km
06530 Ankara, Turkey
{kdincer, emrah, kadriye}@baskent.edu.tr*

Abstract. In this paper, we present a portable, object-oriented, pure Java implementation of the Message-Passing Interface (MPI), called *jmp*. *jmp* is a class library of Java-routines for specifying and coordinating parallel codes. Our pure Java implementation is distinguished from earlier implementation efforts that pervasively use native methods and provide a Java wrapper functionality to some specific traditional MPI implementations. While bringing in a consistent MPI object model suitable for Java, we also follow the standard MPI Application Programming Interface (API) definitions as closely as possible to keep the learning curve short for experienced MPI programmers. We tested the performance of *jmp*, by running a set of benchmark programs written in Java with calls to *jmp* library routines on a cluster of SUN UltraSparc workstations.

1 Introduction

The use of a collection of general-purpose heterogeneous computer systems interconnected by existing networks and support services as a single, logical computational resource has become a widespread approach to low-cost, high-performance parallel computing. These networked computing systems allow individual applications to harness the aggregate power and memory of often largely underutilized powerful, heterogeneous, and well-networked collections of resources available to many users.

Parallel computing on local networked environments has reached a solid level of maturity in recent years. New and considerable gains in processor performance and network capabilities have evolved hand-in-hand with significant developments in new software tools and programming methodologies. MPI is one of the successful tools that facilitate the use of heterogeneous, networked computers to run parallel applications. MPI was proposed as a standard MPI by a committee of vendors, implementors, and users [1, 2].

The introduction of Java by Sun Microsystems has brought another dimension to the high-performance parallel computing efforts on networked computing environments. Java is easy to learn and scalable to complex programming problems. Using Java as the base language hides difficulties of parallel programming, makes the development of large projects easy and keeps them manageable, simplifies the development and testing of parallel programs by enabling a modular, object-oriented approach based on some extensions to the Java API. Although Java was not specifically designed for computationally-intensive numerical applications that are the typical target of highly parallel machines, its widespread popularity and portability has made it an interesting candidate for “high-performance” parallel programming. With its platform-independent execution model, uniform and portable interface to operating system services such as networking and multi-threading, Java has given hope that easy local area high-performance network computing is an achievable goal. Dealing with threads in the distributed message-passing environment implementations is usually complex and error-prone. Java’s language-level support of threads and its object-oriented tendencies

make it a strong candidate for being used to build and program a thread-based, object-oriented, distributed environment.

We here present an object-oriented message-passing Java class library, named *jmpi* that supports all the MPI-1 functionality as well as some critical aspects, such as dynamic process management and thread safety, of MPI-2. *jmpi* combines the advantages of Java with the well established techniques and practices of message-passing parallel processing on computer networks. *jmpi* was built upon the JPVM system that provides most of the functionality required to set up and communicate in a networked environment. This critical implementation decision saved us considerable time and effort in obtaining the final product. JPVM is a PVM [3] library implementation written in Java and it offers some features not found in standard PVM such as thread safety, multiple communication end-points per task, and default-case direct message routing [4].

The rest of this paper is organized as follows. Section 2 summarizes the related efforts in this area and their similarities to and differences from our work. In Section 3, we explain how we built MPI protocols over the underlying JPVM communication layer. Section 4 presents the *jmpi* performance results collected running a set of benchmark programs and compares the *jmpi*'s performance with PVM and JPVM systems. We conclude the paper summarizing the importance of the implementation and by listing the performance results.

2 Related Work

After the introduction of the MPI standard, several portable MPI implementations in traditional languages were built upon established message-passing libraries. Chameleon-based MPICH [5, 6], LAM MPI [7], the Chimp implementation of MPI, and Unify [8] running on top of Parallel Virtual Machine (PVM) are among successful examples.

Among these, Unify is the most closely related one to our work in terms of the implementation strategy. Unify is a public domain software of Mississippi State University which supports a dual-API that permits the communication functions of PVM, MPI, or both message passing libraries to co-exist in the same application program. Both *jmpi* and Unify implementation modifies PVM functions by adding "contexts" required in MPI protocols, but *jmpi* implementation is done in pure Java while Unify was implemented using the C programming language.

Implementation of Object Oriented MPI (OOMPI) [9] has given us clues about a good object-oriented implementation strategy. OOMPI is a C++ class library specification that encapsulates the functionality of MPI into a functional class hierarchy to provide a simple, flexible, and intuitive interface.

There is also a number of other message-passing library implementations that run on top of the Java Virtual Machine. jPVM (previously known as JavaPVM) [10] is an interface written using Java native methods capability which allows Java applications to use the PVM software developed at Oak Ridge National Laboratory. jPVM allows Java applications and existing C, C++ applications to communicate with one another using the PVM API. HPJava [11] is an extension of Java which provides High Performance Fortran-like distributed arrays and some new distributed control constructs to facilitate access to local elements of these arrays. It supports the SPMD programming model where an HPJava program is allowed to make a direct call to MPI or other communication packages.

JavaMPI [12] is a language binding for LAM MPI 6.1. It uses native methods in Java to provide a wrapping shell for the MPI implementation. JMPI [13] is a commercial effort underway at MPI Software Technology, Inc., to develop a message-passing framework and parallel support environment for Java. It targets to build a pure Java version of MPI-2 standard specialized to commercial applications. JMPI is not a completed product at this time, and no detailed information about the progress or status of the work is publicly available.

Some other efforts in the area concentrate on utilizing Java mainly for heterogeneity and portability. The Visper environment incorporates the advantages of Java with techniques of message-passing parallel processing on commodity network [14]. It consists of a console, which provides a GUI for the user to interact with, and control the parallel-processing environment composed of a network of computers. Both the console and the daemon are standalone, multithreaded Java applications. IceT from Emory University focused on bringing together the common and unique attributes of respective programming models, tools and environments associated with Internet programming and parallel, high-performance distributed computing [15]. The IceT group has also done some work on providing a PVM like environment for parallel processing.

3 Implementation Strategy

jmp provides a familiar and proven-effective programming interface that supports a quick learning curve for experienced MPI programmers. An example *jmp* program where each task broadcasts its rank and collects other task's ranks in a table is given in Figure 1. As far as being careful to access MPI methods through instance or class (in case of static methods) names and using specially defined objects when call by reference need to be simulated, *jmp* programs are not much different from those written in other traditional languages.

The 100% Java implementation helps the *jmp* achieve cross-platform portability by benefiting from the standard Java execution environment. An application written in Java is compiled into an architecture-neutral bytecode format, which then executes on a Java Virtual Machine whose purpose is to hide the characteristics of the underlying platform. This feature opens up the possibility of utilizing more types of resources, such as MS-Windows-based or Macintosh machines commonly excluded from networked parallel computing systems. In addition, the Java programming language plays a significant role in providing the desired level of provisional security imposed on local and remote execution of unowned processes.

```
import jmp.*;
public class jmp_example
{
    public static void main(String args[]) {
        int      NUM_PROCS = 8, errors, i, rank, size;
        int[]    table = new int[NUM_PROCS];
        intPtr   rp, sp;
        jmpEnv   e = jmpConst.jmpInit(args);

        e.jmpComm_rank( jmpConst.MPI_COMM_WORLD, rp = new intPtr(rank));
        e.jmpComm_size( jmpConst.MPI_COMM_WORLD, sp = new intPtr(size));
        rank = rp.value(); size = sp.value();

        if (rank == 0) for (int i=0; i < size; i++) table[i] = i;
        e.jmpBcast(table, NUM_PROCS, jmpConst.MPI_INT, 0,
                  jmpConst.MPI_COMM_WORLD );
        /* Check if we have the correct answers */
        for ( i=0; i<size; i++ ) if (table[i] != i) errors++;
        System.out.print("Proc. "+rank );
        if (errors >0) System.out.println( " is done with ERRORS!");
        else System.out.println( " is SUCCESSFUL!" );
        e.jmpFinalize();
    }
}
```

Figure 1. A sample *jmp* program.

The biggest problem with the Java implementation at the time being is that programs written in Java runs (i.e., is interpreted) about 10 times slower than those written in C. This inferior performance is expected to fade away with the fast evolving hardware support, just-in-time compilation and other compiler technologies for Java. Furthermore, the gap between the CPU and communication performance is rapidly increasing, and programs in such a networked-environment will not be CPU-bound in the future. The network latency is the major limiting factor in many scientific and engineering applications thus slower processing rate of programs in Java can be hidden by network latencies.

Our implementation strategy highly depends on the simulation of required functionality of MPI using existing PVM functions. In this section, main features of MPI not supported by PVM are summarized and some implementation issues are described. MPI focuses on the standardization of process communication, synchronization, and group operations. It supports *contexts* and *groups*, and it adds some new functionality not found in PVM such as *thread safety*, *user-defined data types*, *gather/scatters*, *overlapping groups*, and *virtual topologies*. It offers portability and high performance for a wide variety of problems. Since most of the point-to-point communication primitives of *jmp*i and JPVM can be mapped one-to-one by only replacing the places of their arguments, their discussion is omitted here.

3.1 Contexts and Groups

When sending or receiving a message, the process and message identifiers must be specified. A process involved in a communication operation is identified by group and rank with that group (i.e., Process ID=(group, rank)), whereas messages are considered to be labeled by communication context and message tag within that context (i.e., Message ID=(context, tag)).

A *group* is an ordered collection of process identifiers. Each process is identified in its group by its rank – the order in the process list. Groups can act as sets upon union, intersection, and difference operations, also with the member and subset relationships. The group mechanism is implemented in *jmp*i by mapping each JPVM task identifier to a unique group identifier and rank pair and by providing set operations required to manipulate groups. MPI uses a system-defined tag, or context, to divide a communication domain into noninterfering subdomains and to ensure safe communication. The group and context are specified by means of a communicator object in the argument list of the *jmp*i send and receive routines. Communication contexts are carried as part of the message tags in communication primitives. *jmp*i upgraded the receipt selectivity of certain JPVM primitives and extends the message envelope to include the sender's or receiver's rank, message tag, and a communicator.

3.2 Thread Safety

Although MPI 1.0 definition does not imply any work on threads, most MPI implementations included support for threads to improve the level of parallelism of MPI programs. Unfortunately, most thread library implementations are not only incomplete but also unreliable and unsafe to work on. JMPI depends on Java language's portable, uniform thread implementation and achieves thread safety automatically.

3.3 User-Defined Data Types

While PVM can send the implementation language's built-in data types, MPI defines its own basic data types to ensure compatibility among various platforms. *jmp*i allows the user to define her own data types and their associated methods – including explicit *pack* and *unpack*

operations on the data – as class objects. *jmp*i implementation relies on the Java 1.1 Object Serialization [16] to transparently send objects rather than raw data via TCP sockets.

3.4 Collective Communication Operations

MPI supports several high-level collective data movement routines such as gather, scatter, and broadcast. A scatter operation distributes the contents of a process buffer to other processes in the same context, while gather operation is collecting the data received from each process in the same context to a buffer. Since JPVM (and PVM) does not directly support these operations, they are implemented in *jmp*i by communicating with each participating node in a tight loop.

3.5 Virtual Topologies

Virtual Topologies are used to map the structure of data processed to the processors available. A group can be assigned to either a graph topology with arcs connecting communicating processes or a Cartesian topology such as 3D grid structures. Cartesian topology allows shifting data along dimensions. Also we can perform group operations over several subdimensional portions of the Cartesian topology. *jmp*i explicitly do the mappings of coordinates between Cartesian and processor coordinates.

3.6 Setting Up the *jmp*i Environment and Running a Parallel Application

Advantages of using Java can also be seen in setting up a networked computing environment and performing parallel computations on that environment. Traditionally, in a system like PVM one must first install a PVM daemon on all machines to be used in the computation, compile the application binaries for each target architecture, distribute these binaries to all the machines either by explicitly copying them, or by using a shared file system, provide the owner of the computation with the ability to remotely execute PVM daemons and the application binaries. Using daemons is not an unusual solution in MPI implementations. For example, LAM MPI requires the user to start a LAM daemon on each host before starting of the computations. Since application programs and *jmp*i library codes are written in Java we avoid the second step mentioned above. The same code will execute in the same way in all platforms. Given that a jPVM daemon has been started on each target machine, automatic copying of application byte codes to each machine by using special console functions is also considered trivial. In most MPI implementations a fixed set of processes are created at program initialization and one process is created per processor. This is also done automatically by the *jmp*i by communicating with each daemon and forking a process on each machine.

4 Performance Results

Our benchmark tests were performed on a cluster of five SUN UltraSPARC 1 workstations with 167 MHz processors running the Solaris 2.5.1 operating system connected with a 10 Mb/s Ethernet. The Java codes were compiled using SUN's 1.1.5 Java Development Kit and executed on SUN's Java Virtual Machine. We used PVM version 3.4.Beta4 and JPVM version 0.1 in performing the benchmark tests.

The performance of the communication primitives used to send and receive data is crucial in networked computing environments. Similar to the test methodology given in [17], we used a point-to-point communication benchmark program and performed a Ping-Pong test between a pair of connected machines over the local area network connected using PVM, JPVM, and

jmp. For the sake of clarity, we will call the task running on one of these machines as the *master* process, and the other one as the *slave* process. The master process sends messages of varying lengths to a slave process. On receiving the message, the slave echoes the same message back to its master. Table 1 summarizes the round-trip times and effective bandwidths obtained from our tests using PVM, JPVM, and *jmp*, respectively. The large latencies associated with Java-based implementations significantly reduce the effective bandwidth for small messages as seen in Table 1.

Regression analysis of the transmission time allows the calculation of the start-up latency, the asymptotic bandwidth, and average send time per byte while communicating between a pair of machines as illustrated in Table 2.

The results given in Tables 1 and 2 show that we lose an important portion of the available bandwidth in both Java-based message-passing implementations. In addition to the slower interpretation speed of Java programs, dynamic memory allocation required for each MPI object used in the program results in additional wasted time. *jmp* is layered over JPVM and naturally incurs some more overhead (about 10 to 15%) as compared to JPVM due to additional wrapper layer that reformats function arguments for JPVM routines. However, as the message size increases, the effect of this type of implementation details becomes less significant and PVM to *jmp* round-trip time ratio drops sharply.

We also compared the performance of a matrix multiplication program using JPVM and *jmp*. The results are shown in Table 3 and consistent with the above results.

Table 1. Message round trip time and communication bandwidth.

# of Bytes Sent	PVM		JPVM		<i>jmp</i>	
	Time (ms)	Bandwidth (MB/s)	Time (ms)	Bandwidth (MB/s)	Time (ms)	Bandwidth (MB/s)
4 B	0.93	4.3E-3	50.05	70.99E-5	54.63	7.32E-5
1 KB	1.82	5.49E-1	110.81	9.02E-3	124.19	8.05E-3
10 KB	9.70	1.03	153.30	6.52E-2	173.28	5.77E-2
100 KB	89.93	1.11	490.93	2.04E-1	559.62	1.79E-1
1 MB	917.98	1.09	4156.24	2.41E-1	4573.8	2.19E-1

Table 2. Latency, asymptotic bandwidth, and average send-time per byte.

	PVM	JPVM	<i>jmp</i>
Start-up Latency (ms)	0.921	51.84	58.43
Asymptotic Bandwidth(MB/s)	0.76	0.10	0.09
Time per Byte (μ s)	0.435	1.980	2.176

Table 3. Matrix multiplication times.

Matrix Size	JPVM				<i>jmp</i>			
	Multiplication Time (ms)		Total Time (ms)		Multiplication Time (ms)		Total Time (ms)	
	4 tasks	16 tasks	4 tasks	16 tasks	4 tasks	16 tasks	4 tasks	16 tasks
2x2	337.0	1108.0	1604.0	5225.0	384.0	1219.0	1762.0	5760.0
256x256	46260.0	13445.0	48935.0	17453.0	51813.0	15193.0	56069.0	19451.0
1024x1024	805406.0	742718.0	808956.0	746860.0	910112.0	794709.0	1071154.0	820456.0

5 Conclusions

We described the implementation of a message-passing library called *jmpi* that satisfies the MPI-1 specification. This is a first-time effort that entails both a well-designed API that enables easy writing of MPI parallel programs in Java and a pure Java implementation of MPI. We brought a consistent and high-quality MPI object model suitable for Java. Our work is distinguished from several earlier Java wrapper development efforts for legacy message-passing libraries that pervasively use native methods in the implementation.

References

- [1] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," Computer Science Department, Technical Report CS-94-230 and CS-93-214, University of Tennessee, April 1994; also in *International Journal of Supercomputer Applications*, Vol.8, No.3 & 4, pp. 157-416, 1994.
- [2] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, MIT Press, 1995.
- [3] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine, A Users Guide and Tutorial for Network Parallel Computing*, Cambridge, Mass., MIT Press, 1994.
- [4] A.J. Ferrari, "JPVM: Network Parallel Computing in Java," in Proceedings of ACM 1998 Workshop on Java for High-Performance Network Computing, February 1998.
- [5] N. Doss, W. Gropp, E. Lusk, and A. Skjellum, "A Model Implementation of MPI," Technical Report MCS-P393-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1993.
- [6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message-Passing Interface Standard," Technical Report, Argonne National Laboratory, 1996.
- [7] G. Burns, R. Daoud, and J. Vaigl, "LAM: An Open Cluster Environment for MPI," Technical Report, Ohio Supercomputer Center, Columbus, Ohio, 1994.
- [8] F.-C. Cheng, "Unifying the MPI and PVM 3 Systems," Technical Report, Department of Computer Science, Mississippi State University, May 1994.
- [9] Available at <http://www.cse.nd.edu/~lsc/research/oOMPI/overview.html>
- [10] D. Thurman, "JavaPVM," available at <http://www.isye.gatech.edu/chmsr/JavaPVM/>
- [11] B. Carpenter, G. Zhang, G. Fox, X. Li, and Y. Wen, "HPJava: Data Parallel Extensions to Java," in Proceedings of ACM 1998 Workshop on Java for High-Performance Network Computing, February 1998.
- [12] S. Mintchev, "Writing Programs in JavaMPI," Technical Report MAN-CSPE-02, School of Computer Science, University of Westminster, London, UK, October 1997.
- [13] G. Crawford III, Y. Dandass, and A. Skjellum, "The JMPI Commercial Message Passing Environment and Specification: Requirements, Design, Motivations, Strategies, and Target Users," Technical Report, MPI Software Technology, Inc., 1998.
- [14] N. Stankovic and K. Zhang, "Java and Parallel Processing on the Internet," Technical Report, Department of Computing, Macquarie University, NSW 2109, Australia, 1998. Available at <http://btwebsh.macarthur.uws.edu.au/san/webe98/Proceedings/Stankovic/>
- [15] P.A. Gray and V.S. Sunderam, "IceT: Distributed Computing and Java," in Proceedings of ACM 1998 Workshop on Java for High-Performance Network Computing, February 1998. Available at <http://www.mathcs.emory.edu/~gray/abstract7.html>
- [16] "Java Object Serialization Specification," Sun Microsystems, 1997. Available at <ftp://www.javasoft.com/docs/jdk1.1/serial-spec.pdf>
- [17] N. Yalamanchilli and W. Cohen, "Communication Performance of Java Based Parallel Virtual Machines," in Proceedings of ACM 1998 Workshop on Java for High-Performance Network Computing, February 1998.