

# *jmp* and a Performance Instrumentation Analysis and Visualization Tool for *jmp*

**Kivanc Dincer**

Department of Computer Engineering  
Baskent University  
Baglica Campus, Eskisehir Road  
06530 Ankara TURKEY  
Fax: +90 (312) 234-1051  
kdincer@baskent.edu.tr  
<http://www.baskent.edu.tr/~kdincer/>

## **ABSTRACT**

*jmp* is a 100% Java-based implementation of the Message-Passing Interface (MPI-1) standard. *jmp* comes with a consistent MPI object model suitable for Java. Its Application Programming Interface (API) is similar to the standard C bindings of MPI. *jmp* is integrated with a performance instrumentation, analysis, and visualization system called JPVS, that is also implemented in Java. Instrumented *jmp* routines generate execution trace files in Pablo's SDDF format. These trace files are processed by the JPVS and processor- and communication-oriented static and dynamic performance displays are generated to help *jmp* users to observe the behavior of their programs. We give sample displays from the JPVS along with some early performance results of a set of *jmp* benchmark codes on a cluster of SUN UltraSparc workstations.

**Keywords:** MPI, Java, Pablo SDDF, JPVM, performance instrumentation and visualization.

## **1. INTRODUCTION**

We have recently witnessed a strong trend towards building high-performance networked computing environments by exploiting the aggregate power and memory of increasingly powerful, heterogeneous, and often largely underutilized collections of computational resources on existing commodity networks. This has become a low-cost and high-performance alternative to the last decade's custom built distributed-memory parallel computers. Message Passing Interface (MPI) [1, 2] allows users to write parallel programs for such networked computing environments using a relatively simple high-level message-passing model. It takes care of such things as automatically executing the appropriate code on each of the computing nodes involved, keeping track of existing processors, routing and delivering messages. MPI was proposed as a standard message-passing interface by a committee of vendors, implementers, and users [3].

Shortly after the introduction of Java by Sun Microsystems, its widespread popularity and portability have drawn the attention of many research groups that work in the high-performance parallel computing arena. It was proposed that Java could replace traditional languages such as C, Fortran, and C++ commonly used for parallel programming. Some of the reasons behind this proposition were that:

- Java is easy to learn,
- Using Java as the base language hides difficulties of parallel programming,
- Java makes the development of large projects easy and keeps them manageable,
- Java 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 fodder of highly parallel machines, recent developments in just-in-time Java compilers, Java chips, etc. have given hope that this is a temporary situation.

On the other hand, Java also simplifies the job of system designers that develop tools for exploiting the power of networked computing environments. With its platform-independent execution model, uniform and portable interface to operating system services such as networking and multi-threading, Java can be utilized for the solution of many problems in such environments. 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. At the same time, its native method interface allows the legacy codes and message-passing libraries to be exploited.

We here present an object-oriented message-passing Java class library, named *jmp* that supports all the MPI-1 functionality as well as some critical aspects, such as dynamic process management and thread safety, of MPI-2 [4]. *jmp* combines the advantages of Java with the well established techniques and practices of message-passing parallel processing on networked computing environments. Our pure Java implementation is distinguished from earlier implementation efforts that pervasively use native methods and provide Java wrapper functionality to some legacy MPI implementations.

Developing performance analysis and visualization tools for Java-based high-performance computing is another open area that needs to be addressed. While many tools exist for performance diagnosis of parallel programs written in traditional languages, there is no single tool available yet addressing the high-performance Java computing community's needs. We developed a simple, platform-independent tool, called *JPVS* (Java Based Performance Visualization System) that can be

used to instrument *jmp*i-based programs and to determine the performance bottlenecks by visualizing their behavior.

The rest of this paper is organized as follows. Section 2 presents the tradeoffs of implementing a message-passing library in Java and explains how we built MPI protocols of *jmp*i over the underlying JPVM communication layer. The components of the JPVS system used for instrumentation, analysis, and visualization are described in Section 3. Section 4 presents the *jmp*i preliminary performance results collected by running a set of benchmark programs and compares the *jmp*i's performance with PVM and JPVM systems.

```
import jmp.i.*;
public class jmp_i_example
{
    public static void main(String args[]) {
        int      NUM_PROCS = 8, errors, i, rank, size;
        int[]    table = new int[NUM_PROCS];
        intPtr   rp, sp;
        jmp_iEnv e = jmp_iConst.jmp_iInit(args);
        e.jmp_iComm_rank( jmp_iConst.MPI_COMM_WORLD, rp = new intPtr(rank));
        e.jmp_iComm_size( jmp_iConst.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.jmp_iBcast(table, NUM_PROCS, jmp_iConst.MPI_INT, 0,
                    jmp_iConst.MPI_COMM_WORLD );
        for ( i=0; i<size; i++ ) if (table[i] != i) errors++;
        if (errors >0) System.out.println( " Proc. "+rank+ " is done with ERRORS!");
        e.jmp_iFinalize();
    }
}
```

**Figure 1. A sample *jmp*i program.**

## 2. *jmp*i – AN OBJECT-ORIENTED MPI FOR JAVA

Current MPI implementations can be collected under three groups: implementations in traditional languages, Java wrapper implementations where legacy message-passing libraries are called through the native method interface, and pure Java implementations. The last two share the problem of defining a suitable Java API for the original MPI functions.

Chameleon-based MPICH [5, 6], LAM MPI [7], the Chimp implementation of MPI [8], and Unify [9] running on top of PVM are a few of the successful examples of portable MPI implementations in traditional languages.

In the second group there exist University of Westminster's JavaMPI [10] which is a language binding for LAM MPI 6.1 and NPAC's MPIJava that targets to provide an MPI-type API for MPICH. The use of native methods helps to obtain a high-performance communication layer and hides the slower computation speed of Java-based parallel message-passing programs.

*jmp* presented in this paper and JMPI [11] are the only representative examples of the third category. JMPI 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 for commercial applications. JMPI is not a completed product at this time, and detailed information about the progress or status of the work is *not* publicly available.

*jmp* is a class library of Java routines for specifying and coordinating parallel codes. Specification of a natural Java API for MPI is important for providing an easy-to-use interface to users. *jmp* provides a familiar and effective programming interface that supports a quick learning curve for experienced MPI programmers. We adopted the API proposed by JavaMPI with minor changes but also took some lessons from the MPIJava as a well-thought-out alternative Java interface for MPI. An example *jmp* program where each task broadcasts its rank and collects other tasks' ranks in a table is given in Figure 1. As far as being careful in accessing MPI methods through instance or class (in the case of static methods) names and in using specially defined objects when "call by reference" needs 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 (JVM) 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 network 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.

The biggest problem with the Java implementation at the time being is that programs written in Java run (i.e., are 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 a networked environment may not be CPU-bound in the near future. The network latency is the major limiting factor in many scientific and engineering applications thus the slower processing rate of programs in Java can be hidden by network latencies.

*jmp* is built upon the JPVM<sup>T</sup> system [12] 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 Java-based implementation of PVM [13] 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 [14]. JPVM's implementation is more typical of MPI implementations, rather than the existing PVM architectures. For example, JPVM's message-passing implementation is based on communication over TCP sockets, whereas one would expect a UDP-based task-to-task communication to be supported in the PVM environment. Furthermore, instead of PVM's typical daemon-to-daemon routed communications, JPVM uses direct task-to-task message delivery. Combined with some other implementation characteristics not mentioned here, we decided that JPVM could be used to form the underlying communication layer's skeleton in a typical Java-based MPI implementation. Although use of daemons may not seem to be very common in MPI implementations, it is not an unusual solution either. For example, LAM MPI requires the user to start a LAM daemon on each host before starting any computations. 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 *jmp* by communicating with each daemon and forking a process on each machine.

Our implementation strategy highly depends on the simulation of required functionality of MPI using existing PVM functions. We benefited from the implementation of Unify in this respect. Unify is public domain software from 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 *jmp* and Unify implementations modify PVM functions by adding the communication "contexts" for building MPI protocols, but the *jmp* implementation is done in pure Java while Unify was implemented in C. Analyzing the design of Object Oriented MPI (OOMPI) [15] also gave 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.

Some other ideas about *jmp*'s overall design were inspired by the following three research prototypes: The Visper environment incorporates the advantages of Java with techniques of message passing parallel processing on commodity networks [16]. IceT from Emory University brings together the common and unique attributes of respective programming models, tools and environments associated with Internet programming and parallel and high-performance distributed computing [17]. HPJava [18] provides High Performance Fortran like distributed arrays and some new distributed control constructs to facilitate access to local elements of these arrays.

In this section, the main features of MPI not supported by PVM are summarized and then the way we implemented these features using JPVM is briefly mentioned. Since most of the point-to-point communication primitives of *jmp* and JPVM can be mapped one-to-one, their discussion is omitted here. 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*.

## 2.1 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 within 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* 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* send and receive routines. Communication contexts are carried as part of the message tags in communication primitives. *jmp* 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.

---

<sup>†</sup> JPVM should not be confused with jPVM (previously known as JavaPVM) jPVM is an interface written using Java native methods capability which allows Java applications to use the Parallel Virtual Machine (PVM) software developed at Oak Ridge National Laboratory.

## 2.2 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 in MPI programs. Unfortunately, most thread library implementations are not only incomplete but also unreliable and unsafe to work with. *jmp*i depends on Java language's portable, uniform thread implementation and achieves thread safety automatically.

## 2.3 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 as class objects – including explicit *pack* and *unpack* operations on the data. *jmp*i implementation relies on the Java 1.1 Object Serialization [19] to transparently send objects rather than raw data via TCP sockets.

## 2.4 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.

## 2.5. 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 sub-dimensional portions of the Cartesian topology. *jmp*i implements the functions that can explicitly map the coordinates between Cartesian and processor topologies.

## 3. JPVS – A Performance Monitoring, Analysis and Visualization System for *jmp*i

*Performance diagnosis* can be defined as the process of finding and explaining performance problems, which is an important part of parallel programming. JPVS is a platform independent performance instrumentation, analysis, and visualization tool that is integrated with the *jmp*i system and provides assistance to users with respect to performance diagnosis. The instrumented version of the *jmp*i message-passing library generates execution trace data files in Pablo's Self-Defining Data

---

JPVM allows Java applications and existing C, C++ applications to communicate with one another using the PVM API.

Format (SDDF) format. JPVS processes these trace files off-line and generates processor- and communication-oriented static and dynamic, postmortem displays that help the *jmp*i users to observe and analyze the behavior of their parallel programs.

A number of tools were developed over the years that address the performance diagnosis issue on parallel and distributed environments. COMET [20], IPS [21], PAWS [22], TRACEVÝEW [23], and TATOO[24], Xab[25], XPVM[26], HeNCE[27], EASYPVM[28], IPS-2[29], JEWEL[30], AIMS[31, 32], Medea[33], TATOO [24], ParaGraph [34], and Pablo [35, 36] are some of the many that can be mentioned. There also exist some commercial tools targeting specific platforms.

The tools mentioned above give an idea of the factors affecting performance and how to improve it. They help the system designer achieve efficiency in the overall parallel system and guide the user to understand the behavior and performance of the application programs [37]. However, none of these tools can be used in a Java-based parallel-programming environment. Many of the visual performance analysis tools currently available are either not portable at all (i.e., they only address a specific target platform) or were designed to process a specific and restricted data format. Java offers a solution for portability problems. Besides, the pervasive use of Java for parallel programming brings the need for new tools of performance analysis and visualization that are written in Java.

Our work was mostly influenced by the ParaGraph and Pablo systems. ParaGraph [34] is an offline visualization tool with various kinds of displays and that gives an insight into program behavior and interprocess communications. Although ParaGraph is considered to be one of the most informative performance analysis tools, it has a major disadvantage since it can only process data in PICL [16] format. Pablo [35, 36] of the University of Illinois at Urbana-Champaign is a well recognized performance instrumentation and analysis environment designed to organize and visualize information collected from programs executing on parallel machines.

JPVS supports three major tasks: instrumentation, analysis and filtering, and visualization. Each one of these tasks is explained below in detail:

### **3.1 Instrumentation**

The instrumentation system is responsible for collecting the execution data traces on the fly with minimal and accountable invasiveness. Data traces include the calls made, the time spent for each communication routine, or time spent for waiting for the results coming from other processors, etc. JPVS includes a small set of profiling library calls that let the user specify what information is to be collected during execution (instrument), which in turn indirectly determines how trace data is to be represented (view).

Although performance analysis generally requires the knowledge of architecture-specific data semantics, embedding this information in either trace data format or the analysis software modules will preclude cross-platform portability and extensibility. Therefore we use the Pablo SDDF that specifies both the structures of data records and data record instances. We adopted the MPI instrumentation record definitions given in [38]. An SDDF header is composed of a group of record definitions and determines the semantics of data in a trace file. A simple parser interprets the header information to parse the subsequent sequence of tagged data records [37]. For this reason, using SDDF does not restrict the user to a predefined record set, but allows description of general data records [39]. Using SDDF and Java together in our tool is a great advantage for platform independence.

Although we instrument parallel Java programs with *jmpi* message-passing calls, the system has the potential for being used with legacy message-passing codes as far as the SDDF trace files are generated at run time. This allows PVM programs or other MPI implementations written in Java to use our visualization system.

### **3.2 Analysis and Filtering**

As shown in Figure 2, trace records generated from instrumented programs are stored on local disks on each processing node during the execution. These local trace files complement each other since inter-process communication operations involve multiple endpoints. Once the execution is over, these trace files need to be merged on the central visualization server in order to obtain the complete picture. The visualization server reserves a temporary private buffer for each processing node and assigns a separate receiving thread to each buffer (i.e., connection.) The timestamp-ordered events are merged and written to the overall trace file on the server.

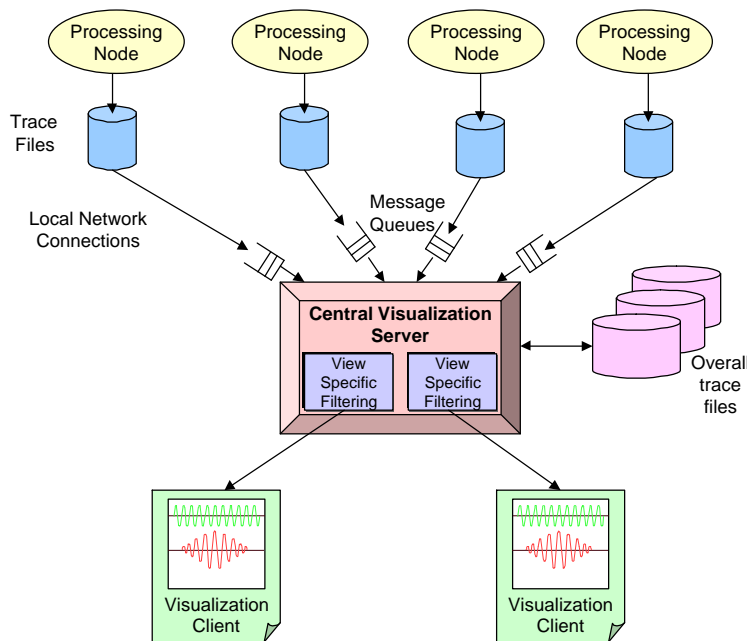
A problem with this client/server architecture is the volume of the data to be transferred between the nodes. Sending local trace files over the network is costly and inefficient. The trace files are transferred to the central server along a network stream by using the Java Object Serialization. Whenever possible – this depends on the type of visualization – the trace data records on local files are filtered before being forwarded to the central server in order to reduce the volume of the data. For example, visualizing the total processor time spent may benefit from local filtering of communication traces, whereas the communication visualization may require collection of all traces.

### **3.3 Visualization**

Visualization of the collected data is a critical element in providing developers with the needed insight into the system under study. The instrumentation system generates enormous amounts of highly

complex execution trace data that is very difficult to understand or interpret. One can imagine the volume of trace data in the case of a large number of computation nodes or long / multiple runs. The human brain is much better able to locate and isolate selected events and activities when presented in graphical form. In order for the programmer to fully understand a parallel program's operation, data from the execution must be analyzed across different levels of observation and from a number of different perspectives. For example, a utilization summary chart can only give the message of poor load balance, but the user detects the exact point of its occurrence by examining the utilization Gantt chart.

JPVS supports processor-oriented and communication-oriented displays. Processor-oriented displays such as processor utilization charts or concurrency profile charts give an insight into workload across processors, the time intervals during which processors work concurrently or processors are *idle* and *overhead* caused by the nature of parallel computation. Communication-oriented displays (communication matrix, message queue length, space-time diagram, etc.) illustrate the overhead caused by inter-process communication and provide a feedback to the user so as to decide how to make performance tuning where the bottleneck occurs, etc. A few sample processor- and communication-oriented displays provided by the JPVS are shown in Figure 3.



**Figure 2. Collection of execution traces and generation of displays through filtering.**

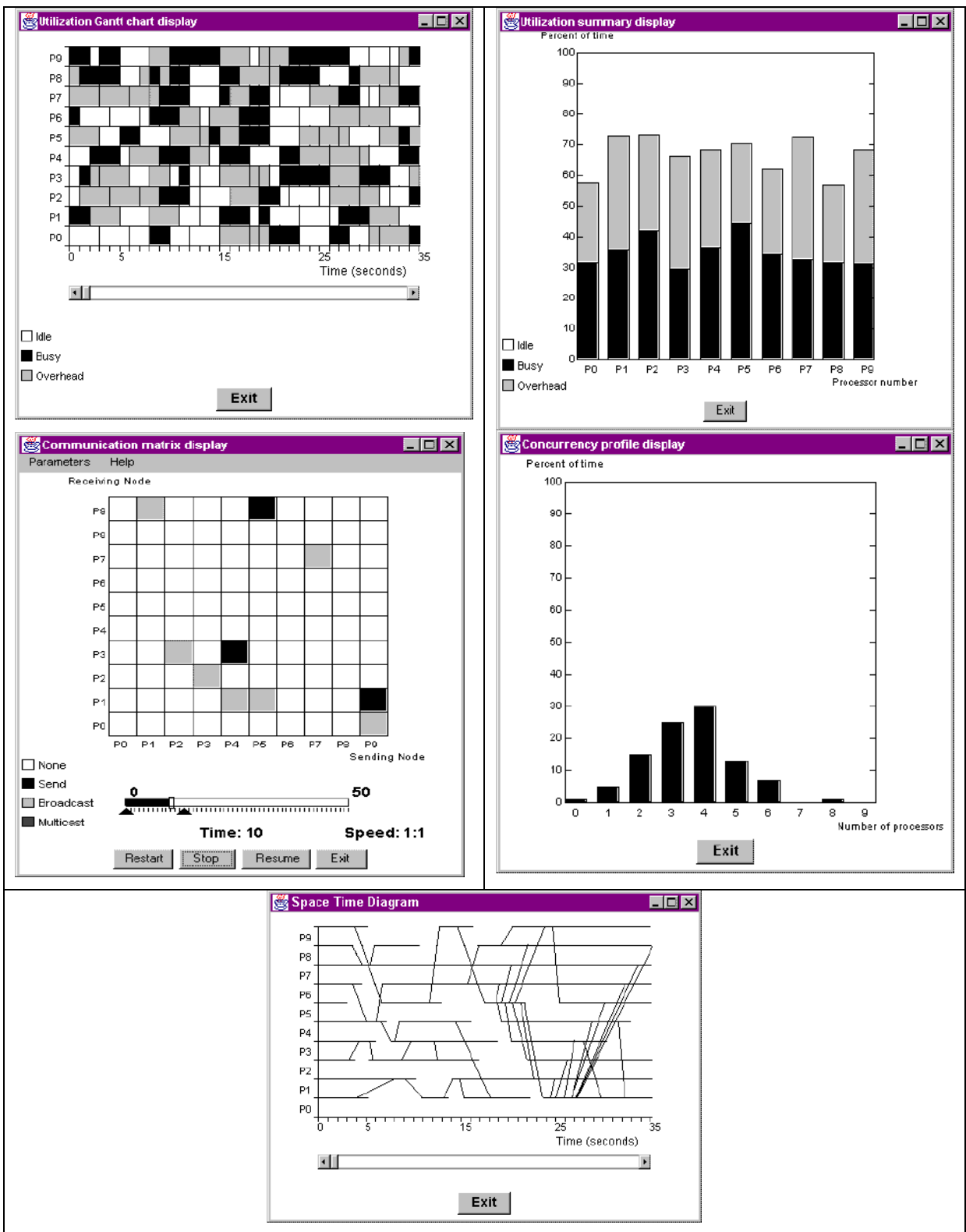


Figure 3. Sample processor- and communication-oriented displays from JPVS.

#### 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 have 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 [40], 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 the master process. 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.

Num.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	4573.86	2.19E-1

**Table 1. Message round trip time and communication bandwidth.**

	PVM	JPVM	<i>jmp</i>
Start-up Latency (ms)	0.921	51.840	58.430
Asymp. Bandwidth (MB/s)	0.760	0.100	0.090
Time per Byte ( $\mu$ s)	0.435	1.980	2.176

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

Matrix Size	JPVM				<i>jmp</i>			
	Mult. Time (ms)		Total Time (ms)		Mult. 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

**Table 3. Matrix Multiplication Times.**

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 in Table 1 and 2 show that we lose an important portion of the available bandwidth on both Java-based message-passing implementations. In addition to the slower interpretation speed of Java programs, the need for dynamic allocation of memory for each MPI object used in the program results in the waste of additional 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, 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*. We can conclude that Java can offer reasonable performance at coarse granularities. The results are shown in Table 3 and consistent with the above results.

#### 4. ACKNOWLEDGMENTS

We would like to thank Dr. Geoffrey C. Fox for his guidance in the initial stages of our work and Mr. Emrah Billur and Ms. Kadriye Ozbas for preparing the screen snapshots and collecting the performance figures.

#### 5. REFERENCES

- 
- [1] W. Gropp, E. Lusk, and A. Skjellum, Using MPI: Portable Parallel Programming with the Message Passing Interface, MIT Press, 1995.

- 
- [2] D. Walker, "The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers", *Parallel Computing*, Vol.20, No.3, pp.657-673, April 1994.
- [3] 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; in *International Journal of Supercomputer Applications*, Vol.8, No.3 & 4, pp. 157-416, 1994.
- [4] Message Passing Interface Forum, "MPI-2: Extensions to the Message-Passing Interface," November 1995.
- [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] Chimp MPI, Edinburgh Parallel Computing Centre. Available from <ftp://ftp.epcc.ed.ac.uk/pub/chimp/>.
- [9] F-C Cheng, "Unifying the MPI and PVM 3 Systems", Technical Report, Department of Computer Science, Mississippi State University, May 1994.
- [10] S. Mintchev, "Writing Programs in JavaMPI", Technical Report MAN-CSPE-02, School of Computer Science, University of Westminster, London, UK, October 1997.
- [11] 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.
- [12] D. Thurman, "JavaPVM," available from <http://www.isye.gatech.edu/chmsr/JavaPVM/>.
- [13] A. Geist, A. Beguelin, J Dongarra, W. Jiang, R. Manček, and V. Sunderam, "PVM: Parallel Virtual Machine, A Users Guide and Tutorial for Network Parallel Computing", Cambridge, Mass., MIT Press. 1994.
- [14] A.J.Ferrari, "JPVM: Network Parallel Computing in Java", in *Proceedings of ACM 1998 Workshop on Java for High-Performance Network Computing*, February, 1998.
- [15] Object Oriented MPI. Available from <http://www.cse.nd.edu/~lsc/research/oOMPI/overview.html>.
- [16] N. Stankovic and K. Zhang, "Java and Parallel Processing on the Internet", Technical Report, Department of Computing, Macquarie University, NSW 2109, Australia, 1998. Available from <http://btwebsh.macarthur.uws.edu.au/san/webe98/Proceedings/Stankovic/www7.html>.

- 
- [17] 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 from <http://www.mathcs.emory.edu/~gray/abstract7.html>.
- [18] B. Carpenter, G. Zhang, G. Fox, X. Li, and Y. Wen, "HPJava: Data Parallel Extensions to Java", in Proc.of ACM 1998 Workshop on Java for High-Performance Network Computing, February, 1998.
- [19] "Java Object Serialization Specification", Sun Microsystems, 1997. Available from <ftp://www.javasoft.com/docs/jdk1.1/serial-spec.pdf>.
- [20] M. Kumar, "Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications", IEEE Transactions on Computers, Vol. 37, pp. 1088-1098, 1988.
- [21] B. P. Miller and C. - Q. Yang, "IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs", in Proceedings of the Seventh Conference on Distributed Memory Computer systems, Vol. 7, pp. 482-489, 1987.
- [22] D. Pease, A. Ghafoor, I. Ahmad, D. L. Andrews, K. Foudil-Bey, T. E. Parpinski, M. Mikki, and M. Zerrouki, "PAWS: A Performance Evaluation Tool for Parallel Computing Systems", IEEE Computer, Vol. 24, No. 1, pp. 18-29. 1991.
- [23] A. D. Malony, D. H. Hammerslag, and D. J. Jablonowski, "TRACEVIEW: A Trace Visualization Tool", IEEE Software, Vol.8, No.5, pp.18-28, 1991.
- [24] R. Borgeest, B. Dimke, O. Hanse, "A Trace Based Performance Evaluation Tool for Parallel Real Time Systems", Parallel Computing, Vol. 21, pp. 551-554, 1995.
- [25] A.L. Begualin, "Xab: A Tool for Monitoring PVM Programs", available from <http://www.cs.cmu.edu/afs/cs.cmu.edu/project/nectar-adamb/web/Xab.html>
- [26] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V.S. Sunderam, "PVM: Parallel Virtual Machine", MIT Press, 1994.
- [27] A. Begualin, J. Dongarra, G.A. Geist, R. Manchek, V.S. Sunderam, "Graphical Development Tools for Network-based Concurrent Supercomputing", in Proc. of Supercomputing 91. Pp. 435-444, Albuquerque, 1991.
- [28] S. Saarinen, "EASYPVM- An Enhanced Subroutine Library for PVM", in Gentzch, W., Harms, U., Proc. Int. Conf. High Performance Computing and Networking, Munich, Germany, April 1994, Lecture Notes in Computer Science 797, pp. 267-271, Springer-Verlag, 1994.
- [29] B.P. Miller, M. Clark, J. Holligsworth, S. Kierstead, S. Lim, and T. Torzewski, "IPS-2: The Second Generation of a Parallel Measurement System", IEEE Transactions on Parallel and Distributed Systems, 1 (1990) 206-217.

- 
- [30] F. Lange, R. Kroeger, and M. Gergeleit, "JEWEL: Design and Implementation of a Distributed Measurement System", *IEEE Transactions on Parallel and Distributed Systems*, 3(6):657-71, November 1992.
- [31] J.C. Yan, "Performance Tuning with AIMS- An Automated Instrumentation and Monitoring System for Multicomputers", *Proceedings of the 27<sup>th</sup> Annual Hawaii International Conference on System Science*, 1994.
- [32] M. Bubak, W. Funika, J. Moscinski, D. Tasak, "Pablo Based Performance Monitoring Tool for PVM Applications."
- [33] M. Calzarossa, L. Massari, A. Merlo, D. Tesserà, "Medea: Parallel Performance Evaluation : The Medea Tool."
- [34] M.T. Heath and J.A. Etheridge, "Visualizing the Performance of Parallel Programs", *IEEE Software* September 1991, pp. 29-39.
- [35] D. A. Reed, R. Aydt, T. M. Madhyastha, R. J. Noe, K. A. Shields, and B. W. Schwartz, "Pablo: An Extensible Performance Analysis Environment for Parallel Systems", Technical Report, Department of Computer Science, University of Illinois, 1992.
- [36] R. J. Noe, "Pablo Instrumentation Environment User's Guide", Technical Report, Department of Computer Science, University of Illinois, October 1996.
- [37] A. Zomaya, ed., *Parallel and Distributed Computing Handbook*, Mc-Graw Hill, New York, 1996.
- [38] H. Simitci, "Pablo MPI Instrumentation User's Guide", Technical Report, University of Illinois at Urbana Champaign, 1996. [<http://www-pablo.cs.uiuc.edu>].
- [39] K. Dincer, "World-Wide Virtual Machine: A Metacomputing Environment Integrating World-Wide Web and High Performance Computing and Communications Technologies," Ph.D. Thesis, Syracuse University, 1997.
- [40] 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.