SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **30**:741–763

An evaluation of Java implementations of message-passing

Nenad Stankovic*,† and Kang Zhang

Department of Computing, Division of ICS, Macquarie University, Sydney, NSW 2109, Australia

SUMMARY

As an objected-oriented programming language and a platform-independent environment, Java has been attracting much attention. However, the trade-off between portability and performance has not spared Java. The initial performance of Java programs has been poor, due to the interpretive nature of the environment. In this paper we present the communication performance results of three different types of message-passing programs: native, Java and native communications, and pure Java. Despite concerns about performance and numerical issues, we believe the obtained results confirm that high-performance parallel computing in Java is possible, as the technology matures and the approach is pragmatic. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: message passing; Java; MPI; PVM; COMMS1; broadcast

INTRODUCTION

When programming for parallel processing, the message-passing model, although less ambitious, has proven more popular than the shared memory. The main advantages are generally better performance, flexibility, and integrability into the existing programming tools and practices. Message-passing libraries like the Message Passing Interface (MPI) [1] and the Parallel Virtual Machine (PVM) [2] provide a common programming interface and a portable program source code across many computer architectures that make up a network or cluster. The library interface is standardized, but multiple architecture, topology specific and performance-tuned implementations of the same library are available. Different implementations employ different approaches, and thus, the performance of each implementation may deviate. When programming in languages like C and FORTRAN it is nevertheless necessary to provide a correct executable for each target architecture.

*Correspondence to: Nenad Stankovic, 2680 Fayette Drive, Apt. 406, Mountain View, CA 94040, U.S.A.
†E-mail: nstankov@ics.mq.edu.au

Received 23 August 1999
Revised 10 January 2000
Accepted 10 January 2000

In 1995 Sun Microsystems introduced a form of machine independent binaries that allow executables to travel over the Internet and execute on any host machine containing the needed interface, referred to as a Java Virtual Machine (JVM) [3]. This new language entitled Java [4] quickly found a home in the entangled Internet Web community, with programmers writing small applications called applets. The fundamental trade-off between portability and performance is well known to programmers of scientific applications for parallel processing. With an interpretive language like the early Java we are looking at an order of magnitude or more slower execution speed. Making Java programs run faster is a challenging task and active area of research. Besides the interpreter, the other approaches to running Java are a hardware implementation, a Hot Spot [5] compiler and a Just-In-Time (JIT) [6] compiler. The most common solution to performance problems is by the use of a JIT that translates Java bytecodes to native instructions, at runtime. The compilation causes some initial slowdown in program execution speed, but significant performance improvements have been observed and reported [7]. The Hot Spot technology is even more promising since it can further optimize bytecodes by performing runtime analysis of the frequency of the code segments execution.

On the other hand, the Java bytecode language is emerging as a software distribution standard, with major software and hardware vendors committed to porting the standard Java run-time environment to their platforms. Being an object-oriented language, Java also has several built-in mechanisms that allow the exploitation of the parallelism that is inherent in scientific computing. Java threads and concurrency constructs that make use of native threads [8] are well suited to shared-memory computers. They also have an important role to play in parallel systems when masking communication and I/O latencies [9]. The performance of a parallel program is also influenced by the speed that data can be exchanged between running processes. The Java networking package provides communication classes based on sockets and Remote Method Invocation (RMI) [10] that can be used for message-passing programming, but are regarded rather low-level and their scalability is questionable. The way the Java I/O has been implemented was found to be a major source of overhead in some benchmarks [11]. The ability to access standard native libraries from Java programs through the Java Native Interface (JNI) [12] is important not only for performance reasons, but also for reusing the wealth of existing C and FORTRAN code with very little cost when writing new applications in Java.

This paper presents a comparison between three different forms: the native, the Java, and the Java-enabled in which programs for parallel processing could be found. Two native libraries, a PVM and an MPI implementation and two Java libraries: the JavaMPI [13] and the jPVM [14] that provide interfaces to these two native libraries have been installed and tested. Also, four pure Java systems: the DOGMA [15], the JPVM [16], the iBus [13], and our own communication library called RTComms. The idea behind these projects is to enable parallel processing in Java based on the mentioned message-passing systems, without modifying the language or the JVM. The tests consisted of the COMMS1 [18] and collective benchmarks collected on different hardware and Java runtime environments. The cost of message passing in Java is analyzed by looking into different steps when sending messages.

MESSAGE-PASSING MODEL

The message-passing programming model represents an application distributed over a collection of processes that communicate problem parameters and results by sending messages. In distributed memory multiprocessors or networked workstations, data are communicated by explicitly invoking

message-passing primitives. When sending a message, a typical message-passing primitive requires a destination address, an identification tag, a content and (depending on the implementation and the programming language) its length and data type. To receive a message, the receiving process provides the senders address and tag to match against, and possibly a buffer with a maximum length. Message-passing primitives often come as blocking (synchronous) and non-blocking (asynchronous). They operate either between a pair of processes or on a group of processes. For example, the broadcast primitive replicates a message onto a group of processes, while the barrier synchronizes a group.

Performance of a message-passing primitive is measured in units of time or *bandwidth* and expressed as *bytes-per-second*. The time for a small or zero-length message to travel through the channel from the sender to the receiver is known as *latency*. It generally depends upon the speed of the signal through the media and any software overhead in sending or receiving the message. Small messages play an important role in evaluating synchronization cost and determining optimal granularity of parallelism. Large messages are primarily affected by the available network bandwidth, with the media maximum usually being approached. However, choosing only two numbers to represent the performance of a network can be misleading. For better comparison and understanding of the behavior of a message-passing environment, communication times are often being drawn as function of message length [19].

For two directly connected processes, message-passing time is usually a linear function of message size. A number of factors that can affect the time to communicate a message can be identified. For networks with a more complicated topology, a per-hop delay may increase the message-passing time. Buffer alignment on word, cache-line or page may also affect performance, while message lengths that are powers of two or cache-line size may provide better performance than smaller lengths. Context-switch times may contribute to delays for small messages. Touching all the pages of the buffers can reduce virtual memory effects, by swapping them into the memory prior to invoking a message-passing primitive. Among other effects, it has been observed that exchanging *worm up* and synchronization messages before collecting timing data can eliminate some first-time effects. The aggregate bandwidth of the network, the amount of concurrency, and congestion management may be issues.

In a system like Java, many of these effects and techniques to eliminate or alleviate them are out of control. On the other hand, Java, due to its architecture and execution mode, has introduced some new issues to be aware of and deal with. If running as a stream of bytecodes, the virtual machine branches out of the switch statement to perform an action, which often results in a cache miss. Run time (e.g. JIT) compilation may solve this problem, but due to a limited time when compiling, the compiler may not produce an optimal native code. The introduction of a garbage collector, and the usage of threads in Java-based message-passing libraries further complicate the situation, as thread creation times and scheduling become issues. For example, Java does not provide a mechanism to discriminate against blocked threads upon notification. The programmer can resume either only one thread at random or all of them, also in no particular order. Efficiency of I/O operations very much depends on the implementation and optimizations in the virtual machine.

THREE TYPES OF IMPLEMENTATIONS

Four key aspects of Java bytecode-based software distribution motivate our interest in Java. First, Java adds security [20] to the distribution of software, providing added benefits over more common languages like C and C++. Second, the JVM delivers the environment that executes Java bytecode

programs on a wide variety of architectures, allowing development of a truly portable software base. Third, Java contains many features that are vital for the success of any universal language. The existence of threads, for example, greatly simplifies the overlapping of I/O and computation, but their usage, if not prudent, can prove costly. Last, due to the nature of interpreted languages, Java executables run slower than their compiled counterparts in other languages. Different solutions are being investigated to overcome this problem.

In our study, we investigate and evaluate pure Java and Java-enabled systems for message-passing parallel programming. To evaluate the results, we compare them against the two popular native systems that should define the range of values we aim for in Java. The JNI provides the bridge for calling native methods from Java. This feature is desirable not only to access legacy libraries but also to speed up performance and enable interoperability with legacy systems. Based on the native code dependability, we can divide the systems of interest into Java-enabled systems (e.g. JavaMPI, jPVM), and pure Java (e.g. JPVM, iBus, RTComms). While implemented in Java, DOGMA relies on native code (when possible) for better communication performance.

MPI and PVM

The message-passing model has become the paradigm of choice for parallel processing. Although there are many variations, the basic concept is well understood and has been widely exploited for the SPMD type of programs on parallel computers and networks of workstations (NOW). MPI has become an emerging standard for implementing SPMD, and since the release of the initial MPI specification [21], several MPI implementations have been made publicly available. The standard is primarily concerned with message-passing issues and performance, and leaves the question of process creation open. The LAM [22] programming environment and development system that has been used in the tests comes, nevertheless, with a complete set of tools to compile, run and debug parallel programs.

Similar to MPI, PVM is used to transform a network of computers into a metacomputing environment. Although less rich in different modes and communication primitives, a PVM implementation is a self-contained system that offers the utilities to spawn and control processes. A PVM program is implemented as a set of processes that may join and leave the application, thus promoting a dynamic processing environment.

JavaMPI and jPVM

It is believed that for Java to establish itself in scientific programming, the interoperability with native legacy software through the JNI is very important. This is also important for performance reasons, especially when no method apart from the interpreting is used to execute the bytecodes. In principle, binding of a native library to Java can be accomplished either by dynamically linking the library to the JVM, or by linking the library to the object code generated by a stand-alone Java compiler. Complications stem from the fact that Java implements strong type safety, has no pointer arithmetic, and Java data formats are different from those found in C. Therefore, a native method interface allows C functions to access Java data and perform a format conversion if necessary. In binding a native library to Java portability problems may arise. The JNI was not part of the original specification, and incompatible interfaces exist from different vendors.

In an effort to combine the advantages of the new features offered by Java, and yet not to sacrifice to much of the performance, access to communication libraries like the MPI and PVM has been enabled. The JavaMPI [13] binding allows Java programs to use a native MPI library, which was LAM in our case. It comes with the Java-to-C interface generator (JCI) that takes as input a C header file and generates a Java native method declaration for each exported function. Thus generated stubs convert the arguments into a form understandable to the corresponding C functions. However, the conversion comes at a cost, which is obvious from our benchmarks. The JavaMPI consists of more than 120 functions, and so far was the most complete and best supported among similar products. For example, the HP Java project has been revived recently, with publicly available mpiJava [23] being an object-oriented Java interface to the standard MPI. It supports MPICH [24] and Sun HPC-MPI [25] on Solaris, and WMPI 1.1 [26] on Windows NT. We believe the performance should be similar to JavaMPI, as both systems take the same approach. (The older version of mpiJava would not run on the benchmark system, and it also required a patch to be applied to the MPICH.) The jPVM [14] interface is similar to JavaMPI in its approach, but provides interoperability between Java and PVM.

DOGMA

DOGMA [15] is a Java environment designed to run parallel programs on networks of workstations and supercomputers (IBM SP/2). The team has worked hard to improve the bad image of Java as being too slow for scientific processing with some encouraging results. Regarding communication aspects, DOGMA follows MPI by implementing a pure Java library called MPIJ that is based on the C++ bindings to the standard MPI as much as possible. MPIJ implements a large subset of MPI functionality, but misses on virtual topologies and user-defined data-types. Objects must be manually serialized before communicated as a stream of bytes, what probably also requires a user defined protocol. To achieve better performance, native types are first marshaled into a Java byte array, and then sent over a TCP/IP channel. For some architectures (e.g. MS Windows) the conversion is performed by a native library, which increases performance by 40% [15]. The library uses persistent communication channels that remain active as long as the program executes. While this feature improves performance, it does not scale.

JPVM

JPVM [16] is a simple but, nevertheless, popular prototype that partially implements a PVM-like environment in Java. The interface supported by the library is similar to the C and FORTRAN interface to PVM, but with syntax and semantics modifications that match better the programming style of Java. It also enhances PVM by the novel features such as thread safety, multiple communication endpoints per task, and direct message routing by default. The communication library is based on the Java Object Serialization [27] to send user-defined data, and persistent socket channels each of which is serviced by its own thread and maintains its own send and receive message lists. The send is asynchronous, while the receive blocks. An asynchronous receive is also available. As in PVM, the programmer must manually pack the data into a buffer and unpack from it. The implementation model is flat since each send (and receive) operation comprises streaming of three integers, two character arrays and one object.

Not all PVM communication primitives have been implemented, most notably there is no barrier synchronization primitive. Being entirely implemented in Java, JPVM does not support interoperability

with the standard PVM. The JAPE [28] project enhanced the JPVM functionally by implementing the barrier and reduction primitives, and performance-wise by improving task creation and message delivery. JPVM was also used as the basis to build an MPI-like system in Java called jmpi [29].

iBus

iBus [17] is an object-oriented, pure Java middleware aimed at supporting intranet applications (e.g. groupware, multimedia) and as such is not a message-passing library *per se*. However, it can serve as a basis to build one. The main strength lies in the development of location independent applications, with no single point of failure and no background services, based on TCP/IP and reliable IP multicast. The communication model in iBus is asynchronous (i.e. push) or synchronous (i.e. pull). For this benchmark, we have used the push since the pull (re)uses the same channel for sending and receiving. This has required a simple synchronization mechanism that blocks the sending of a new message before a reply is received for a previous one. Rather than providing wrapper methods for sending and receiving the benchmark used the standard mechanism of creating a posting with a data object that contained a specified number of bytes, and pushing it down a TCP stack. The following code explains the mechanism:

```
Stack stack = new Stack("TCP"); // TCP quality of service
iBusURL url = iBusURLFactory.create(...); // create channel
Data data = new Data(); // wrapper object
Posting posting = new Posting(1);
data.data = new byte[1000]; // allocate user data
posting.setLength(1); // constrain posting to 1 object only
posting.setObject(data); // add data to posting
stack.push(url,posting); // send posting
```

Upon receipt of a message, a callback method gets invoked that passes the posting in. The usage of a data object as the wrapper when sending native types resembles the mechanism found in RTComms, as described below. The COMMS1 test program has been run between two dedicated nodes without a logical-to-physical address resolution.

The iBus version used in the tests was the last nonlicensed release made by Softwired. More recent versions might be obtained free of charge for research (check the Softwired web-site), but they have a different API, based on Java beans.

RTComms

RTComms is a communication class that follows the MPI standard, but also takes advantage of the features like the function overloading and the Java Object Serialization to simplify the implementation and programming. The RTComms class represents the API to our thread-safe communication library. It is a part of our metacomputing environment called Visper [30]. The communication architecture is presented in Figure 1.

As in iBus, to help with data marshaling, all the message-passing methods take serializable objects as arguments, and consequently messages consist of serializable Java objects that are communicated via sockets. When serializing data, the programmer does not have to define the data type or the array

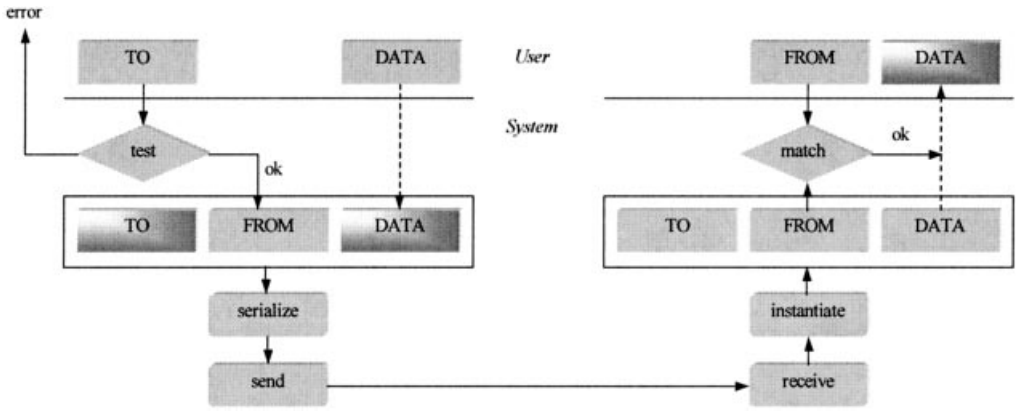

Figure 1. Communication architecture.

size. Being properties of the data, that is taken care of by the serialization mechanism itself. These features simplify the signature of MPI primitives in our implementation, and reduce the possibility of programming errors. As in MPI, each message has a tag, a process ID and a group to which the sending or receiving process belongs. These three attributes are combined into an envelope object of the abstract system VData class. When sending data, a RTDataSend object is used, and when receiving a RTDataRecv is used. Both classes are derived from VData, and define the methods to match against tag, group or process ID. All the primitives that send data take another argument that represents the contents. For example, the blocking send is defined as:

```
RTComms rtc = new RTComms();
RTDataSend envp = new RTDataSend(tag,toProcID,group);
Data data = new Data(new byte[1000]);
rtc.Send(envp,data);
```

and the matching blocking receive:

```
RTComms rtc = new RTComms();
RTDataRecv envp = new RTDataRecv(tag,fromProcID,group);
byte[] b = ((Data)rtc.Recv(envp)).data;
```

The implementation supports blocking (synchronous) and non-blocking point-to-point and collective messaging, together with synchronization capabilities in a raw and trace mode. At the moment, there is no support for virtual topologies. Synchronous and asynchronous primitives can be combined together, at both ends of a communication channel. To perform arithmetic operations, as required by MPI_Reduce, we use the Reflection API, to look inside the object for native types. For example, to sum up a similar array of data as above at *root* from a *group* of nodes, the reduce operation is defined as:

```
RTDataReduce envp = new RTDataReduce(root,group);
Data result = (Data)rtc.Reduce(envp,data,RTReduceOp.ID_SUM);
```

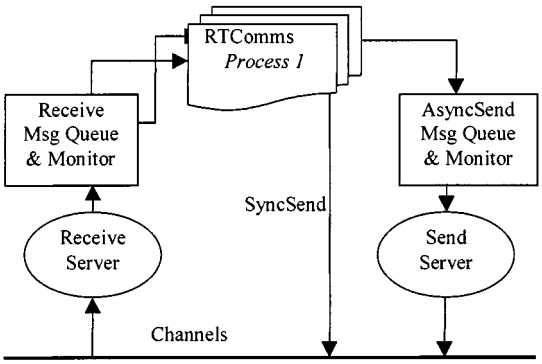

Figure 2. Internal architecture.

RTComms does not use persistent communication channels, which explains the selection of the push model in iBus to compare against.

Internally (Figure 2), the communication library is structured as a symmetrical composite entity that consists of two threads, two monitors and an API class, the RTComms, at the top. All the individual components, on each side, are grouped upon the *producer–consumer* pattern. The receive server (i.e. thread) creates and services a server socket. Upon a connection request, a new channel is passed to the receive monitor that extracts the envelope and the data, appends them to the message queue, and notifies all the blocked processes that new data have arrived. The monitor also implements the methods to check and get synchronously and asynchronously data from the queue, and to match them against an envelope. Once a match was found, the RTComms method returns the data. When sending a message, a synchronous send makes a direct socket call and blocks. An asynchronous send appends a message to the send monitor message queue. The send server gets notified and sends the message. Messages can be canceled, and checked for completion synchronously or asynchronously. An error status is raised upon failure. For better response, the receive server runs at the highest priority, while the send server runs at the same priority as the API thread (i.e. process) that has notified it, for fairness.

BENCHMARKING

To send a message from one process to another the following steps are required:

- Network address resolution,
- Data marshaling, and
- Data transfer.

In message-passing programming, the network address resolution involves the conversion of a logical process identifier with respect to a communication group into an actual IP address and port number. The data marshaling involves the conversion of the data from the local host format into the network format,

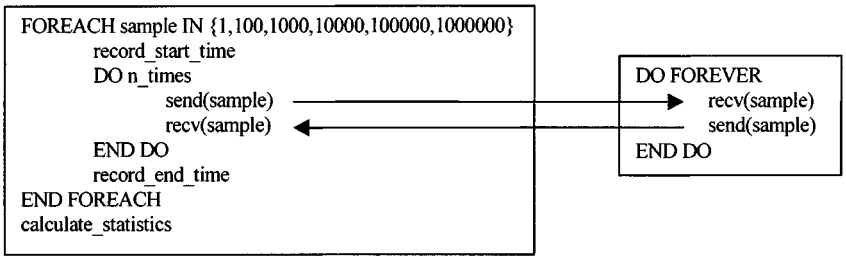

Figure 3. COMMS1 pseudo-code.

and *vice-versa*. The data transfer comprises the establishing of a reliable communication channel. These three steps were included in the benchmarking results presented below, except for the iBus test program. The iBus times, in the tables below, do not include the network address resolution. Assuming that the information is available locally, it should not cause much overhead, especially in a JIT or HotSpot environment which is of main interest. In the case of RTComms, each message consists of two objects (i.e. message envelope and content) which requires two serialize/deserialize calls per message. In the iBus case, only the message content was serialized, as only one object was pushed on the stack. To make a fair comparison with the Java systems, performance in all the tests was measured at the application level.

COMMS1

The COMMS1 [18] benchmark is used to measure the basic communication properties of a message-passing computer by measuring the end-to-end delay. Also known as the *pingpong* benchmark, it requires two processes, where each process acts as a sender or a receiver interchangeably. The receiving process simply echoes back whatever was sent, and the sending process measures roundtrip time. Times are collected outside the repetition loop as defined by the pseudo-code in Figure 3.

It is assumed that the time to send a message is equal to the time to receive a reply. The communication times for blocking primitives were measured for messages of various lengths, i.e. 1, 100, 1000, 10 000, 100 000, and 1 000 000 bytes.

Collective

In this benchmark, we focus on two collective communications primitives: the broadcast and the barrier, since they are standard in most parallel-programming environments. To measure broadcast performance, in the benchmark we vary the message size and the number of involved processes. The algorithm is presented in Figure 4.

To measure barrier performance only the number of processes varies, as no data are sent. This benchmark is also known as SYNCH1 [31] and it measures the time to execute a barrier synchronization primitive as a function of the number of participating processes, or as the number


```
FOREACH sample IN {1,100,1000,10000,100000,1000000}
        record_start_time
        DO n_times
                broadcast(sample)
        END DO
        record_end_time
END FOREACH
calculate_statistics
```

Figure 4. Broadcast pseudo-code.

Table I. Hardware.

Vendor	OS	Architecture	RAM(MB)	CPU(MHz)
Sun	Solaris 2.5	Ultra 2	256	$168 * 2$
HP	HP-UX B.10.20	A 9000/700	512	180
Compaq	NT 4.00.1381	Pentium II	64	200
Micron	NT 4.00.1381	Pentium II	256	400

of executions per second. It is expected that the time to execute a barrier does not increase too fast with the number of processes.

Collective primitives can be implemented with point-to-point primitives. The sender communicates with each participating node in a tight loop. The problem with this approach is that it is inefficient for high latency environments, and waist bandwidth as it does not scale up well. On the other hand, to broadcast reliably over a network requires programming with timeouts that is not convenient at the API level.

The environment

The environment consisted of three different hardware architectures. Only the Java benchmarks were performed on all the three, while the native and Java-to-native tests were performed only on Sun. We have used the following software in our tests:

- LAM 6.2b (University of Notre Dame) compiled with gcc2.7.2 on UltraSparc,
- PVM 3.3 compiled with gcc2.7.2 on UltraSparc,
- JavaMPI and jPVM as downloaded of the Internet,
- iBus version 0.5,
- JDK 1.1.6 without JIT from Sun Microsystems for UltraSparc/Solaris,
- JDK/JRE 1.2.1 from Sun Microsystems for PC/NT with JIT and HotSpot compilers, and
- HP-UX Java C.01.15.05 with JIT.

Table II. LAM and PVM times (ms).

Length	LAM	PVM	JavaMPI	jPVM
1	0.358	0.58	1.08	0.95
100	0.540	0.78	0.78	1.10
1000	2.043	2.36	2.28	2.65
10 000	9.987	10.4	10.2	10.8
100 000	91.99	95.8	97.1	97.6
1 000 000	922.3	966	957.7	985.1

The hardware is defined in Table I. The network was a 10 Mbps Ethernet. For each of the tests, we have used a pair of workstations of the same architecture. The cross-architecture results were dominated by the slower component, and can be deducted from the tables.

THE RESULTS

To compare communication systems several metrics are required. The study presents and compares the results for the mentioned libraries. The time-per-byte and latency values were calculated by linear regression. A summary of the results based on the tables below is presented in a series of graphs.

COMMS1

The pingpong test was performed on all the mentioned libraries and architectures. While the results for the native systems exhibit linear increase as the messages get larger, the Java systems are much more erratic, due to the problems described above (see section on Message-Passing Model).

Table II presents the results collected for the native and Java-to-native systems. They were all collected on the Sun boxes, which mans that no JIT was used for Java. Nevertheless, the impact is not severe and is diminishing as the messages were getting larger, due to the increasing network overheads. The overhead in the Java systems is also caused by the argument conversion in the wrappers.

Table III presents the results collected for the RTComms library on the three different architectures and Table IV for the iBus. The iBus JIT results compared to the HotSpot on PC were mixed, e.g. for 1 and 100 bytes it took 85 ms with JIT and for 100 000 and 1 000 000 it took 135 and 1407 ms, respectively.

The performance results for DOGMA in Table V follow the pattern observed above. It is nevertheless interesting to notice the effect due to the native code enhancement built into the library on small messages on PC, in particular, which is quite dramatic. On HP and Sun, however, the marshaling of the data was performed only in Java. Another important factor is the persistent communication channel between the nodes.

The JPVM approach is to a certain extent similar to the one found in RTComms, as both libraries use object serialization. In the JPVM case, only one object, the data buffer gets serialized. The JPVM

Table III. RTComms times (ms).

Length	Sun(−jit)	HP(+jit)	PC(+HotSpot)
1	60.83	13.2	6.7
100	61.73	13.3	6.8
1000	65.91	14.9	8.4
10 000	88.8	102.7	16.6
100 000	179.5	205.5	140.9
1 000 000	1201	1084	1427

Table IV. iBus times (ms).

Length	Sun(−jit)	HP(+jit)	PC(+HotSpot)
1	40.3	99.4	75
100	54.5	100	95
1000	49.2	107.6	100
10 000	68.2	200.6	20
100 000	170.6	203.4	130
1 000 000	1261	1141	1317

Table V. DOGMA times (ms).

Length	Sun(−jit)	HP(+jit)	PC(+HotSpot)
1	1.04	0.843	0.864
100	1.11	0.863	0.825
1000	2.63	2.343	2.655
10 000	12.4	20.11	12.52
100 000	108.4	162.4	126.4
1 000 000	1080	1021	1274

uses persistent communication channels, which explains the better latency time on Sun (Table VI). However, as it relies heavily on threads, scheduling becomes an issue. This is particularly important on HP, where threads were rather lazy. The PC times are harder to explain, as they do not fall into the observed pattern of behavior, being exacerbated by the extremely high latency value (Figure 5).

From the presented results it can be observed that for the pure Java libraries the performance is much slower for small messages, which means that the latency values are high, as is obvious from Figures 5 and 6. All the figures present only the more interesting results, for brevity. The results in Figure 6 are normalized to Sun LAM, as the better DOGMA value is due to a better hardware. As the messages

Table VI. JPVM times (ms).

Length	Sun(−jit)	HP(+jit)	PC(+jit)
1	27.2	102	138.25
100	25.4	100	109.4
1000	25.0	100	109.35
10 000	21.0	25.2	18
100 000	123	116.6	120
1 000 000	1108	1011	1258

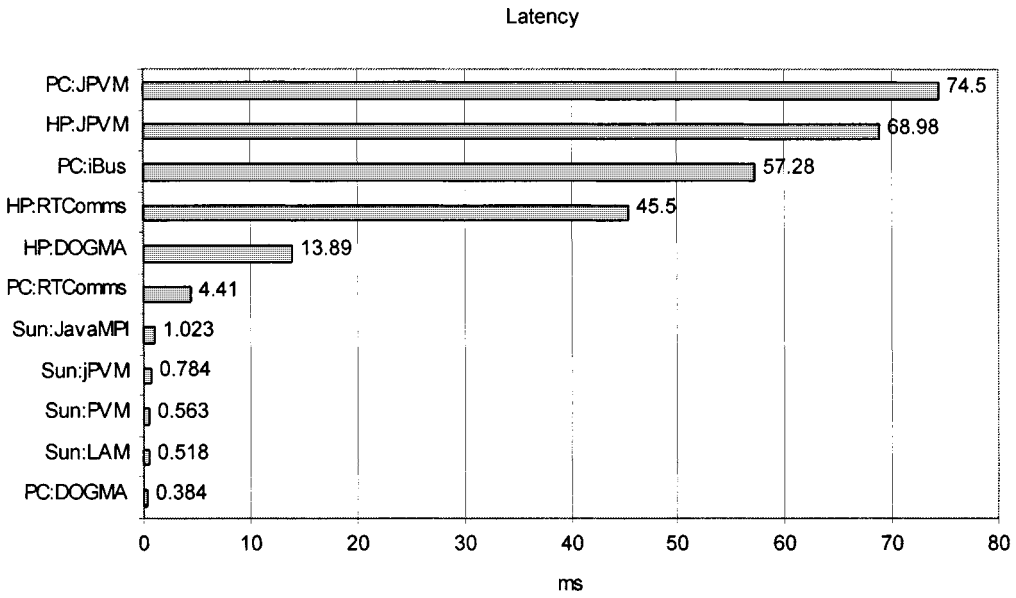

Figure 5. Latency (ms).

were getting larger, the differences in roundtrip times were getting smaller. Consequently, the time per byte values are close (Figures 7 and 8). It is encouraging, nevertheless, that performance could be improved significantly by employing a Java-to-native compilation technique. The approach taken by DOGMA also speaks in favor of Java, since they have pinpointed and implemented a relatively simple solution to significantly boost I/O performance. Rather than having a legacy message-passing system as in JavaMPI, they have only added native code to convert Java native types to bytes before sending them through a channel. If the standard Java would provide such functionality (at the moment

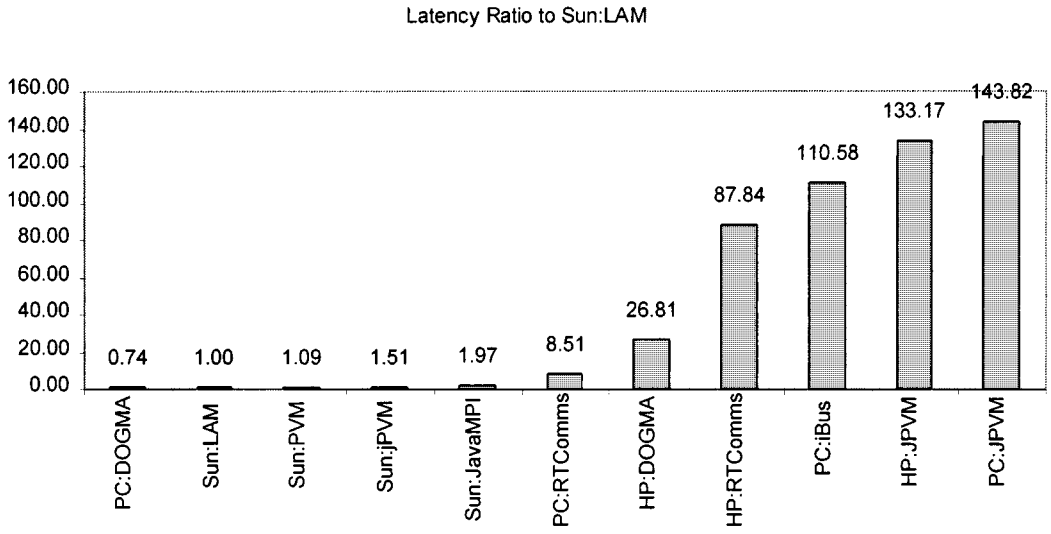

Figure 6. Latency ratio.

there is only the *System.arraycopy* method), that code would not be needed. However, the problem of serializing objects would still remain, since objects are not linear entities like arrays, but are in fact graphs of types.

The channel throughput or bandwidth is the rate at which the network can deliver data. As a result of the high latency values, the effective bandwidth is reduced for small messages in the RTComms and iBus case compared to the native based libraries, by one to two orders of magnitude. The calculated bandwidth values are presented in Figure 9. Table VII shows the bandwidth for LAM, PVM, JavaMPI and jPVM. The values are high and represent the results aimed for in Java systems.

Table VIII shows a selection of bandwidth values for iBus, JPVM and RTComms. With respect to the native systems, the results are an order of magnitude or more inferior for small messages. The iBus bandwidth values are very close to the RTComms values. In the iBus case, however, only a message content was serialized, as only one object was pushed on the iBus stack. The presented results are nevertheless encouraging. For small messages, the JPVM bandwidth values are one order of magnitude inferior to the RTComms values.

Table IX shows the bandwidth values collected for DOGMA. For small messages they are one to two orders of magnitude better than the iBus and RTComms values. The difference diminishes as messages get larger.

If we look at the calculated bandwidth across the whole range of messages, the picture is different, and the results remain within a 30% range that favors pure Java approach (Figure 9).

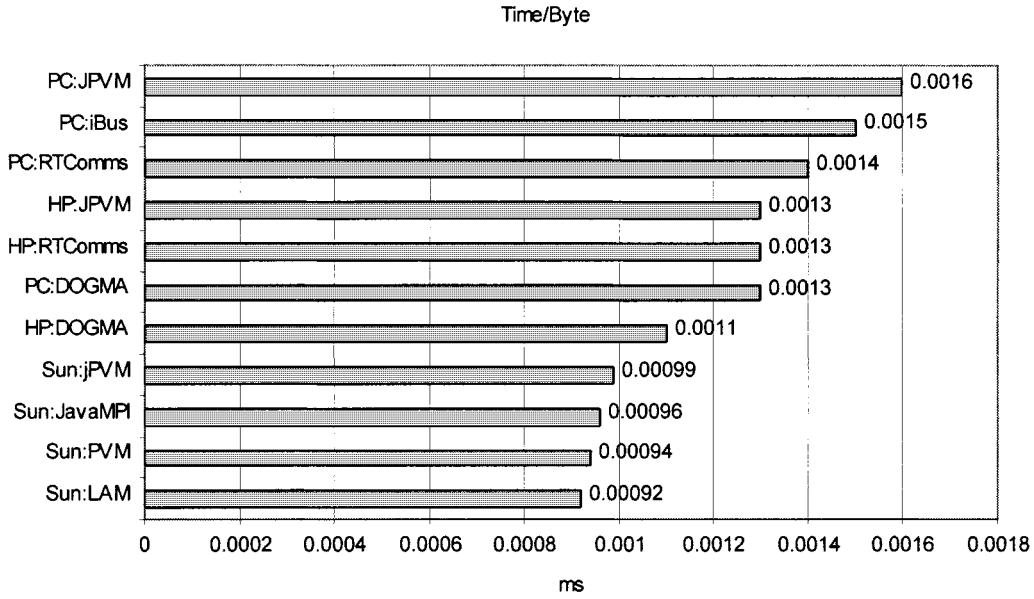

Figure 7. Time/byte (ms).

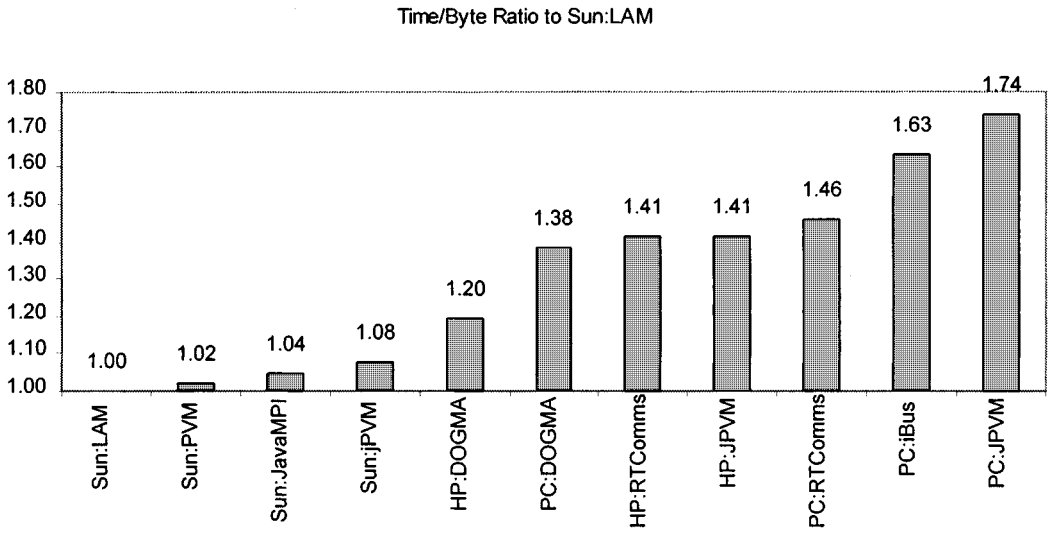

Figure 8. Time/byte ratio.

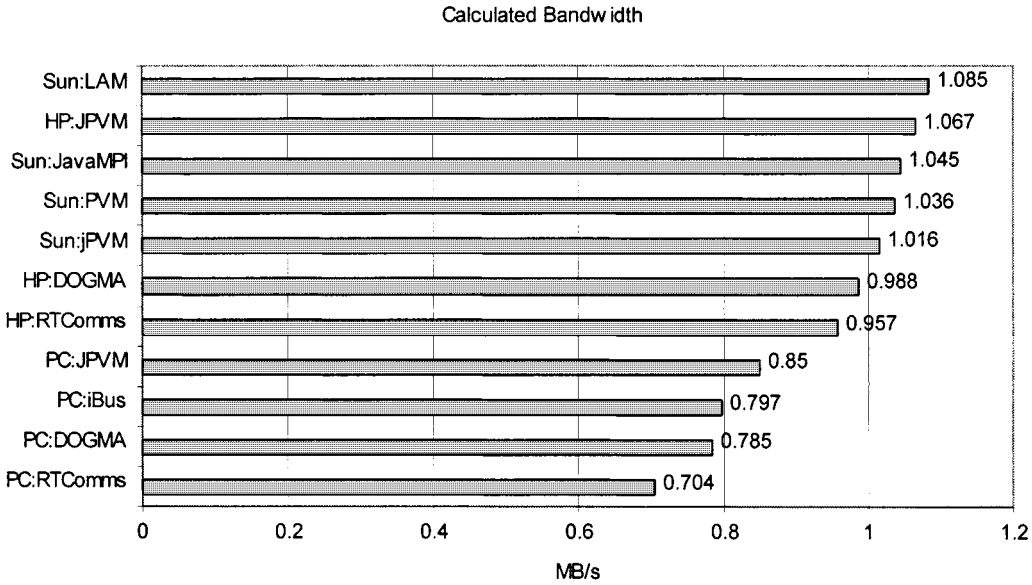

Figure 9. Calculated bandwidth (MB/s).

Table VII. LAM and PVM bandwidth (MB/s).

Length	LAM	PVM	JavaMPI	jPVM
1	0.003	0.002	9.2e−4	0.001
100	0.19	0.13	0.13	0.091
1000	0.49	0.423	0.44	0.378
10 000	1.00	0.96	0.98	0.926
100 000	1.09	1.043	1.03	1.025
1 000 000	1.08	1.035	1.04	1.015

Barrier

The barrier synchronization performance is dominated by startup latency, as no user-defined data are sent. It requires that each process in a group first sends a message to the root process, and waits blocked before the root replies back. If based on point-to-point primitives, the lower the latency, the more efficient the implementation. A smart library should not perform a network call when delivering locally, even though operating systems often provide optimizations in such cases. Therefore, in Figure 10 most results for only one node fall at or near 0 ms. For the Java libraries, 0 indicates a value that was below timer resolution.

Table VIII. iBus, JPVM and RTComms bandwidth (MB/s).

Length	PC: iBus	PC:JPVM	PC:RTComms
1	1.3e−5	7.23e−6	1.5e−4
100	0.001	9.14e−4	0.015
1000	0.01	0.009	0.119
10 000	0.5	0.556	0.602
100 000	0.769	0.831	0.710
1 000 000	0.759	0.794	0.701

Table IX. DOGMA bandwidth (MB/s).

Length	Sun	HP	PC
1	9.66e-4	0.001	0.0012
100	0.09	0.116	0.1212
1000	0.38	0.427	0.377
10 000	0.80	0.497	0.799
100 000	0.922	0.616	0.791
1 000 000	0.926	0.979	0.785

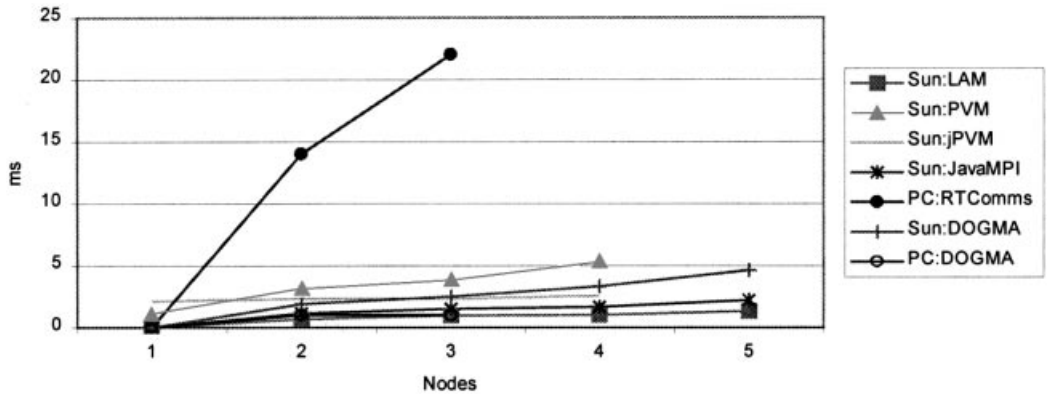

Figure 10. Barrier times (ms).

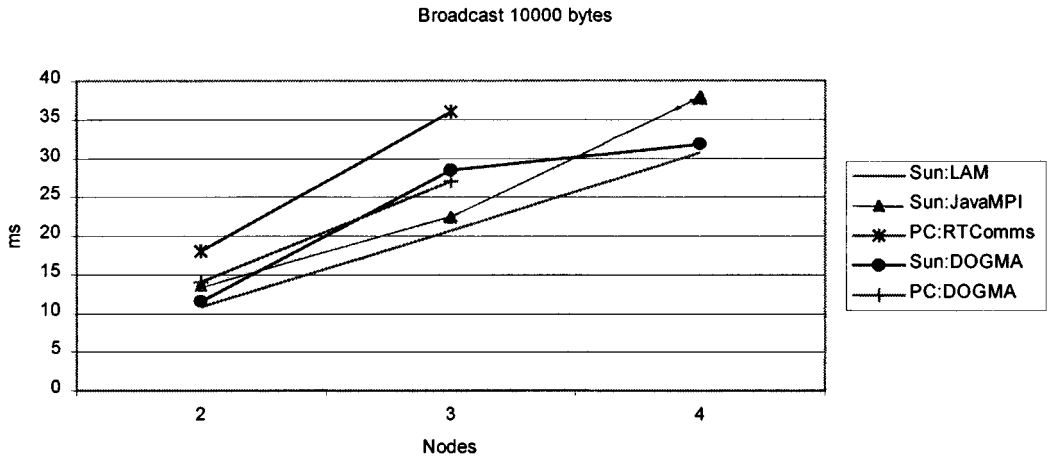

Figure 11. Broadcast times (ms).

In RTComms there are no empty messages, and that's why the results are inferior. The communication library does not discriminate against message data, except for finding a match against the envelope. Therefore the receiving thread always reads two arguments from a channel, and inserts them into a message list.

Broadcast

All the systems under investigation use software-based approaches to cater for broadcast functionality. Performance of collective communications depends on the performance of the underlying point-to-point primitives. That is obvious from the obtained results that show that the bandwidth is inversely proportional to the number of involved nodes. Figure 11 presents a selection of the times to broadcast 10 000 bytes for *PC:RTComms*, *Sun:LAM*, *Sun:JavaMPI*, *PC:RTComms*, and *PC:DOGMA*. The displayed times were measured at the root process. For small messages, the results were dominated by latency.

In Java, the situation with collective communications is further complicated when object serialization is used directly on top of a socket. If realized as a sequence of point-to-point channels, this means that for each channel, the object gets serialized anew. Regarding the efficiency of the object serialization in Java at present, that can prove costly, not only for large objects (see the *PC:RTComms* results). A potential solution would be to use native broadcast, but since it is unreliable it must be timer driven, and therefore not always suitable for programming.

DISCUSSION

While the majority of the results as presented are not very much in favor of Java, it is encouraging to see the improvement as the result of JIT compilation. Given a system like Java, it can be expected that

Table X. Socket creation time (ms).

Sun		HP		PC	
Min	Max	Min	Max	Min	Max
2	7	1	8	0	30

sending a small message does not come cheap. The question remains, however, what is the cause and cost at different stages when sending a message, and how to improve it.

As the starting point, the three steps when sending messages are used, as mentioned in the introduction to the Benchmarking section. Assuming that the address information is stored locally, this is just a local call that returns an *InetAddress* object based on a group name and a host ID number. Even with no JIT, this operation takes less than a millisecond and is thus below the Java's timer resolution, and can be excluded as a potential problem. Then, a socket object is created to establish the communication peers. This is a standard Java library class and its efficiency depends on the JVM. Table X summarizes the spread of values on the benchmark network when creating a socket. It is important to notice that the spread, although large, is in reality much smaller since the values were concentrated at the minimum, and the maximum values were rare. The 0 value indicates that the actual time was below resolution. By comparing these values with those in Table III it is clear that they do not dominate the roundtrip time. The reason for the best ratio being found on Sun is that the socket creation process consists mainly of native methods, and therefore is least intrusive to an interpretive environment. The presented analysis was done on RTComms.

Once the socket has been created, the data could be serialized, i.e. written into the channel. When serialized, a Java object is turned into an array of bytes that includes the native attributes and other objects it refers to. The serialization mechanism is general and performs automatic marshaling and demarshaling by inspecting the object at runtime. At the receiver, these bytes are converted back into a deep copy by restoring the complete object graph. To avoid repetition and infinite loops, the serialization mechanism keeps a hash table of the already visited objects. The serialization time presented in Table XI represents the writing of two objects, as defined by the *RTComms.Send* method. The method blocks before the serialized data get moved out of the process address space. The HP results are surprising, as one would expect the values to be closer to the PC values for small messages. Similar results are obtained even when the process ping-pongs locally.

Finally, we look at the receiving side of the communication channel where the serialized object is restored to its original state, but in a new process. The RTComms library is implemented as a multithreaded entity, following the producer–consumer pattern (Figure 2). The sending thread makes a direct call when sending data, while the receiver is blocked at a monitor. The receive server, after receiving a message, adds the data to a monitor, which notifies all the blocked threads that a new data have arrived. Table XII summarizes the results, that show that for small messages most of the time is spent while deserializing the input from the channel. There are two objects to deserialize: the envelope and the content, as described in section on RTComms. The presented values do not include any time

Table XI. Serialization time (ms).

	Sun		HP		PC	
Length	Min	Max	Min	Max	Min	Max
1	2	3	2	2	0	0
100	2	3	2	3	0	0
1000	2	3	2	3	0	0
10 000	62	70	2	9	10	20
100 000	145	199	209	218	90	101
1 000 000	1131	1213	1017	1062	922	981

Table XII. Deserialization time (ms).

	Sun		HP		PC	
Length	Min	Max	Min	Max	Min	Max
1	54	55	12	25	0	10
100	53	55	11	16	10	10
1000	53	54	12	13	10	10
10 000	115	161	77	98	10	20
100 000	161	280	263	310	110	141
1 000 000	1155	1264	1035	1165	1312	1422

when the receiver was blocked waiting for input. Therefore, no thread synchronization affected the obtained values. Rather, this is just the cost of making two *VObjectInputStream.readObject* calls. *VObjectInputStream* is derived from *java.io.ObjectInputStreams*, and knows how to rebuild objects that belong to classes from other class repositories than those defined in the *CLASSPATH* environment variable, by contacting a customized class loader.

There are two ways to improve on these results. One is to keep the communication channel open, and the other is to use native types rather than objects whenever possible. A problem when keeping channels open, is that this solution does not scale, as there is a limit on the number of file descriptors that can concurrently be in use. Specific to Java, we have observed that for large messages (e.g. 1 000 000 bytes) the flushing of the channel by the sender does not always guarantee that the object will be successfully restored at the receiver. If not, the socket blocks indefinitely. It is not clear, however, why is this the case. With native types, it is required to have a loop that reads in the whole message, and the following code is used:

```
for(int k=0;k<length;k+=is.read(buffer,k,length-k));
```

where:

Table XIII. Native types (ms).

Length	Sun	HP	PC
1	0.279	0.6	0.0
100	0.442	0.5	0.5
1000	1.91	2.05	2.0
10 000	11.21	10.25	12.5
100 000	105.1	101.56	111.2
1 000 000	1035.7	932.85	1160.7

Table XIV. Object times (ms).

Length	Sun	HP	PC
1	1.37	1.79	1.35
100	1.42	1.36	0.9
1000	2.95	2.87	2.44
10 000	11.96	16.2	43.4
100 000	103.6	102	134.7
1 000 000	1032.5	966	1415

k is the offset into the byte array **buffer**
length is the message length in bytes
is is the java.io.DataInputStream.

Regarding the sending of native types rather than objects, we have also written and performed a COMMS1 test of sending an array of bytes of variable length over a persistent TCP channel, the results of which are presented in Table XIII. The test includes only the communication cost, as no threads scheduling or logical-to-physical address resolution were included in the code. The results closely follow those in Table V for DOGMA, as they represent a similar approach.

Table XIV is similar to Table XIII but uses a wrapper object as a native array holder. The communication channel was persistent. We can observe that as messages get larger, the difference in time gets smaller. In both cases, the array was allocated for each different length only once, so the presented times depend mostly on the JVM and the network.

The results in Table XIII are, for small messages, very much in favor of sending native types rather than objects, to improve performance. From the system design perspective, by sending objects rather than native types we can achieve a simpler API and we stay more in line with other mechanisms like the RMI [10]. Neither the programmer nor the API needs to explicitly specify at any point in time what is the data type they deal with or how to go about it when sending messages. As we aim for an object-oriented environment that is open in Java terms, that approach serves us well.

A similar set of tests has been reported in [32], for the PVM suite of tools. The results are surprising for Java given the fact that JIT compilation was enabled. For comparison, the tests were performed also on integers. As expected, they proved to be more expensive due to a host-to-network and network-to-host conversion. A similar setup was also used to benchmark jmpi [29], apparently without JIT compilation. Some observations about the slow performance of the object serialization mechanism combined with RMI can be found in [33].

CONCLUSION

In this paper we have presented our own and related work on high performance message-passing computation in Java, and compared it to the standard native libraries such as MPI and PVM. Versions of the COMMS1 and collective benchmarks were written and tested for each of the libraries. The results were, as expected, in favor of the native systems, with the Java-enabled ones being a clear second. The iBus and JPVM results on PC are probably due to the internal architecture and thread scheduling. Due to a rather old version of Solaris a JIT or HotSpot compiler could not be used on Sun. The HotSpot technology has not significantly improved performance over the JIT. It is encouraging that runtime compilation has improved RTComms performance by almost an order of magnitude on PC for small messages, even though that network subdomain appeared slower than the Unix one. We believe the Sun and JIT results would even closer match the jPVM and JavaMPI results.

The rationale behind combining Java and native code has been in utilizing the standard communication and mathematical libraries, and harvesting their better performance, while gaining on features and flexibility provided by Java. The question remains concerning what is the advantage of using Java in a manner that may add more complexity than C or C++ due to its features, such as the lack of pointer arithmetic and incompatible data types. Due to the different presentation, converting data from Java to native presentation and *vice-versa* comes at a cost and is complicated due to the Java strict type checking and absence of pointer arithmetic. This makes the JavaMPI data model rather confusing compared to the genuine MPI. This explains the reasons behind the Visper/RTComms approach that aims for a pure Java parallel-processing environment. Further improvements are, nevertheless, necessary.

REFERENCES

1. Gropp W, Lusk E, Skjellum A. *Using MPI, Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1994.
2. Geist AG, Beguelin A, Dongarra JJ, Jiang W, Manchek R, Sunderam VS. PVM 3 User's Guide and Reference Manual. *Technical Report ORNL/TM-12187*, Oak Ridge National Laboratory, 1993.
3. Lindholm T, Yellin F. *The Java Virtual Machine Specification* (*Sunsoft Java Series*) (2nd edn). Addison Wesley Developers Press, 1999.
4. Campione M, Walrath K. *The Java Tutorial Second Edition: Object-Oriented Programming for the Internet*. Addison Wesley, 1998.
5. Sun Microsystems, Inc. The Java HOTSPOT Performance Engine Architecture. *White Paper*, April 1999.
6. Neffenger J. The Volano Report: Which Java platform is fastest, most scalable? A Java World exclusive!. http://www.javaworld.com/javaworld/jw-03-1999.
7. Mangione C. Just in time for Java vs. C++. *NC World* 1998; **7**(2).
8. Lea D. *Concurrent Programming in Java, Second Edition: Design Principles and Patterns* (*The Java Series*). Addison-Wesley, 1999.

9. Foster I, Tuecke S. Enabling technologies for web-based ubiquitous supercomputing. *Proceedings of the 5th IEEE Symposium on High Performance Distributed Computing*, http://www.globus.org/nexus/nexusjava.html.
10. Sun Microsystems, Inc. *Java Remote Method Invocation Specification, Revision 1.42*, October 1997.
11. Hsieh C-HA, Conte MT, Johnson TL, Gyllenhaal JC, Hwu W-MW. A study of the cache and branche performance issues with running Java and current hardware platforms. *Proceedings of IEEE CompCon'97*, San Jose, CA, 1997; 211–216.
12. Liang S. *Java Native Interface: Programmer's Guide and Specification* (*The Java Series*). Addison-Wesley, 1999.
13. JavaMPI: a Java Binding for MPI. http://perun.hscs.wmin.ac.uk/.
14. Thurman DA. jPVM. http://www.isye.gatech.edu/chmsr/jPVM.
15. DOGMA: Distributed Object Group Metacomputing Architecture. http://zodiac.cs.byu.edu/DOGMA [September 1998].
16. Ferrari A. jPVM. http://www.cs.virginia.edu/~ajf2j/jpvm.html.
17. SoftWired AG. *IBus Programmer's Manual*, Version 0.5, August 20, 1998. http://www.softwired.ch/ibus.htm.
18. Dongarra JJ, Meuer H-W, Strohmaier E. The 1994 TOP500 Report. http://www.top500/org/.
19. Dongarra JJ, Dunigan, T. Message-passing performance of various computers. *Technical Report CS-95-299*, University of Tennessee, Knoxville, TN, May 1996.
20. Gong L. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation* (*The Java Series*). Addison-Wesley, 1999.
21. MPI Forum. MPI: A Message-Passing Interface Standard, Version 1.0. http://www.mcs.anl.gov/mpi [5 March 1994].
22. Ohio LAM Version 6.1. MPI Primer/Developing with LAM. http://www.mpi.nd.edu/lam [1996].
23. Carpenter B, Getov V, Judd G, Skjellum T, Fox G. MPI for Java: position document and draft API specification. *Technical Report JGF-TR-03*, Java Grande Forum, November 1998.
24. MPICH-A Portable Implementation of MPI. http://www.mcs.anl.gov/mpi/mpich.
25. Sun Microsystems, Inc. Sun MPI 3.0 Guide. *Revision A*, November 1997.
26. WMPI-Win32 Message Passing Interface. http://dsg.dei.uc.pt/wmpi.
27. Sun Microsystems, Inc. Java object serialization specification. http://java.sun.com/. [November 1998].
28. Imasaki K. JAPE: the Jave parallel environment. *ISCOPE'97, The 1997 International Scientific Computing in Object-Oriented Parallel Environments Conference, Marina del Rey, CA*. http://www.acl.lanl.gov/iscope97.
29. Dincer K. A ubiquitous message passing interface implementation in Java: jmpi. http://www.baskent.edu.tr/~kdincer.
30. Stankovic N, Zhang K. Java and network parallel processing. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings 5th European PVM/MPI Users' Group Meeting, LNCS* 1497, Alexandrov V, Dongarra JJ (eds.). Springer Verlag: Liverpool, UK, 1998; 239–246.
31. Hockey R, Berry M. Public international benchmarks for parallel computers. *PARKBENCH Committee: Report 1*, February, 1997.
32. Yalamanchilli N, Cohen W. Communication performance of Java based parallel virtual machines. *ACM 1998 Workshop on Java for High-Performance Network Computing*, Palo Alto, CA. http://www.cs.ucsb.edu/conferences/java98/.
33. Caromel D, Vayssiere J. A Java framework for seamless sequential, multi-threaded and distributed programming. *ACM Workshop Java for High-Performance Network Computing*, Stanford University, Palo Alto, CA, 1998; 141–150.

