MPJ: Enabling Parallel Simulations in Java

Mark Baker, Bryan Carpenter and Aamir Shafi

Distributed Systems Group, University of Portsmouth
Mark.Baker@computer.org, B.Carpenter@omii.ac.uk,
Aamir.Shafi@port.ac.uk

Abstract. Java is emerging as a popular platform for scientific and engineering simulations. Its success can be attributed to its portable nature, good performance, and inherent support for security, threads, objects, and visualisation (graphics). In this paper, we present a message passing system called MPJ, which is an implementation of MPI in pure Java. MPJ supports the parallelisation of large-scale Java simulations, on distributed and parallel systems.

1 Introduction

Java is emerging as a popular platform for scientific and engineering simulations. Its "write once, run anywhere" philosophy allows Java applications to be executed on almost all popular platforms. Being an object oriented language, it supports polymorphism, inheritance, and so on, which are natural to represent real-world objects in simulations. Recent improvements in JIT (Just In Time) compilers, which translate bytecode into the native machine code at runtime, have improved the performance of Java applications. Our earlier study [1] showed that networked Java applications can compete with the performance of their C counterparts. Also, it supports multi-threading and provides simple primitives like `wait()` and `notify()` that can be used for synchronised access to shared resources. More recently, the JDK (Java Development Kit) has been extended to provide semaphores and atomic variables. The standard JDK contains packages like javax.swing, java.awt, and java.awt.event, which facilitate the programming of visualisation tools for computer simulations. There also exists an extension API, Java 3D which can be used to build 3D visualisations.

Security is an important concern for Internet-enabled applications — especially in the context of "The Grid". The JDK comes with three APIs that form the basis of security. JCE (Java Cryptography Extension) allows public and private key generation. JAAS (Java Authentication and Authorization Service) provide controlled access to JVM resources. Lastly, JSSE (Java Secure Socket Extension) ensures secure Internet communications based on encrypted links between two parties.

Exploitation of Java for simulation projects has been ongoing for some years. Fox and Furmanski [4] in one of the earlier works in this area identified the potential of Java. The authors argue that in terms of low-level parallelism, Java's role is to provide wrappers to the native MPI implementations. JWarp [2] is a

Java library for discrete-event parallel simulations, which builds its own communication infrastructure based on Java Remote Method Invocation (RMI). Teo et al [15] discuss a discrete event simulation tool based on conservative algorithms; this system uses JavaSpaces for communications, which is an abstraction of distributed shared memory. MONARC [10] is a simulation framework for large scale computing resources. It has been deployed on an inter-continental testbed to verify simulation results with some success. CartaBlanca [14], from Los Alamos National Lab, is a general purpose non-linear solver environment for physics computations on non-linear grids. It employs an object-oriented component design, and is pure Java. These projects suggest that Java has already made its mark on a range of projects involved in parallel simulations.

We note that most of these tools use RMI for communication. One project uses Jini as the communication medium, which in turn uses RMI. In general, middleware technologies like RMI are used for Internet communications, but there is a need for an efficient MPI like communications library that can be used for communications in Java based parallel applications. In this paper, we present MPJ, which is an MPI (Message Passing Interface) [9] implementation written in pure Java. MPJ provides a high-level communication library and runtime infrastructure that can be used to develop and execute parallel Java simulations.

2 Motivation and Aims

There have been various efforts over the last decade to develop a Java messaging system based on the MPI standard. These systems typically follow one of three approaches: use a JNI (Java Native Interface) to interact with a underlying MPI; implement Java messaging from scratch using the likes of RMI; or implement communications on lower-level Java Sockets API.

A drawback of the first approach (using JNI) is that it does not comply with the "write once run anywhere" philosophy of Java—also, there are some performance overhead in JNI, especially for large messages, due to copying of the data from the JVM heap onto the system buffer [16]. On the other hand, JNI has the advantage that it allows Java to access specialised communication hardware like Myrinet or Infiniband. The second approach, that uses RMI, is not really appropriate as the RMI package is designed for client-server interaction rather than message passing between peers. The use of low-level "pure" Java communications based on Java sockets is the third approach. Our initial benchmarks of the Java New I/O device [1] demonstrate that it is possible to achieve performance close to C implementations of the MPI on Fast Ethernet. However, with this approach it is not possible to take advantage of specialised hardware.

Experience gained with these implementations suggests that there is no "one size fits all" approach. The reason for this is that applications implemented on top of Java messaging systems can have different requirements. For some, the main concern could be portability, while for others high-bandwidth and low-latency could be the most important requirement. Portability and high-

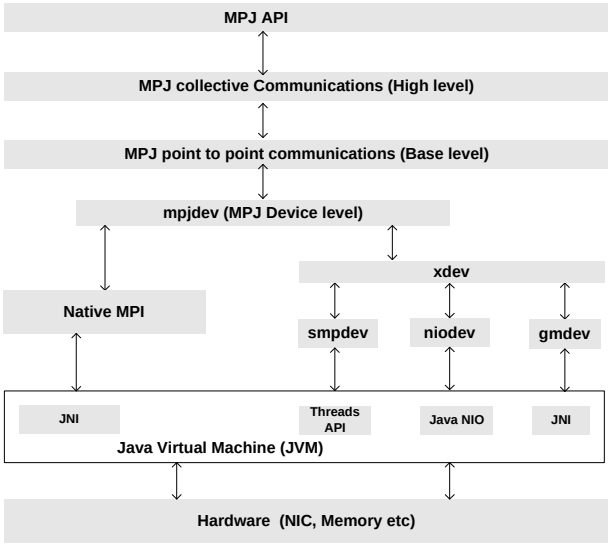

Fig. 1. MPJ design

performance are often contradictory requirements. High performance can be achieved by making use of the specialised communication hardware, but only at the cost of compromising portability that Java offers. The challenging issue is how to manage these contradictory requirements.

To address this issue, we are implementing Message Passing in Java (MPJ) [3] following a layered architecture based on an idea of device drivers. The idea is analogous to UNIX device drivers. We are implementing a Java NIO (New I/O) [13] based device, a shared memory device, a GM [12] device, and a native MPI device. The ability to swap devices at runtime helps mitigate the contradictory requirements of the applications. In addition, we are implementing a runtime system that bootstraps MPJ processes over a collection of machines connected by a network. Though the runtime system is not part of the MPI specifications, it is an essential element of MPJ if we wish to execute processes across various platforms.

3 MPJ design

MPJ has a layered design to allow incremental development, and provide the capability to update and swap layers in or out as needed. Figure 1 shows a layered view of the messaging system that shows MPJ high and base level, *MPJ device* level, and *xdev* level.

The high and base level rely on the *MPJ device* [8] and the `xdev` level for actual communications and interaction with the underlying networking hardware. The API for MPJ is based on the Java bindings defined in [3].

```
public static Device newInstance(String dev)
public ProcessID[] init(String[] args)
public ProcessID id()
public void finish()
public mpjdev.Request isend(mpjbuf.Buffer buf, ProcessID destID, int tag, int context)
public void send(mpjbuf.Buffer buf, ProcessID destID, int tag, int context)
public mpjdev.Request issend(mpjbuf.Buffer buf, ProcessID destID, int tag, int context)
public void ssend(mpjbuf.Buffer buf, ProcessID destID, int tag, int context)
public mpjdev.Status recv(mpjbuf.Buffer buf, ProcessID srcID, int tag, int context)
public mpjdev.Request irecv(mpjbuf.Buffer buf, ProcessID srcID, int tag, int context,
                            mpjdev.Status status)
public mpjdev.Status probe(ProcessID srcID, int tag, int context)
public mpjdev.Status iprobe(ProcessID srcID, int tag, int context)
```

Table 1. The `xdev` API

We envisage two implementations of the MPJ Device level. The first implementation provides JNI wrappers to the native MPI implementations. The second implementation uses the lower level device called `xdev` to provide access to Java sockets, shared memory, or specialized communication libraries. `xdev` is not needed by wrapper implementation because native MPI is responsible for selecting and switching different communication protocols.

The design aim for this API is to minimize the code written for `xdev`, so that new protocols can be written quickly and with less effort. Figure 1 also shows three implementations of `xdev`. smpdev is the shared memory device for MPJ, `niodev` is based on the Java NIO, whereas; gmdev provides JNI wrappers to the GM communications library. For the first time, we present the API for `xdev` in Table 1.

4 MPJ implementation

One of the fundamental differences between Java and C is the lack of pointers in Java. Using C, it is possible to use `void *` to point to any basic datatype and/or structures. This is an important characteristic, which forms the basis for efficient messaging. Thus, the first step in developing MPJ was to develop a buffering API that can be used to store all Java basic datatypes and objects. A reference to this buffer can be passed to the lower level communication devices, which transfers or copies the bytes to the destination.

4.1 MPJ buffering API

The derived datatypes and explicit packing/unpacking for datatypes is achieved through `mpjbuf` (MPJ buffering API), which supports methods like `write`/`read`, `gather`/`scatter`, and `strided-gather`/`strided-scatter`. A buffer object consists of one or more sections that may contain different datatypes. There are two primary sections for a buffer; the static section of the buffer contains Java primitive datatypes, whereas; the dynamic section of the buffer contains Java objects. More datails about `mpjbuf` API can be found in [1].

4.2 Point to point communications

MPJ provides blocking and non-blocking point-to-point communications that could be used to send arrays of basic Java datatypes as well as objects. Also, MPJ provides four modes of send, which have been defined in the MPI specification document.

4.3 Communicators, groups, and contexts

MPI provides higher level abstractions to create parallel libraries, which include communicators, groups, and contexts. Communicators along with groups provide process naming; each process in MPI is identified by its rank. The context, which is an attribute of a communicator provides a safe communication universe—it can be thought of an additional tag on the messages. Also, we have implemented collective communications on top of point-to-point communications, which are useful in writing parallel applications.

4.4 The communication protocols

Implementations of `xdev` device encapsulate various communication protocols. Currently, `niodev`, which is an implementation of `xdev` using Java NIO implements two communication protocols. We discuss each of them briefly here.

The *eager-send protocol* is used by `niodev` for communicating small message, typically less than 128 Kbytes. This protocol works on the assumption that the receiver has got an unlimited device level memory where it can store messages. There is no exchange of control messages before doing the actual data transmission, thus minimizing the overhead of control messages that may dominate the total communication time of small messages. Whenever a send method is called, the sender writes the message data into the socket channel assuming that the receiver will handle it. At the receiving side, there can be more than one scenario, depending on whether a matching receive method is posted by the user or not. If a matching receive method is posted, the message is copied onto the user specified memory. However, if a matching receive is not posted, then the message is stored in a temporary buffer. Later, when the receive is posted, it copies the message from temporary message to the user specified memory

The *rendezvous protocol* is used for large messages, typically greater than 128 Kbytes. There is an exchange of messages between the sender and the receiver before the actual transmission of the data payload. The overhead of this exchange of messages is negligible in terms of the overall communication cost of large messages.

4.5 The runtime infrastructure

MPJ relies on a runtime system that is used to bootstrap MPJ processes over a collection of computers. The runtime consists of starter and daemon modules that execute as the native OS service by using Java Service Wrapper [6] project.

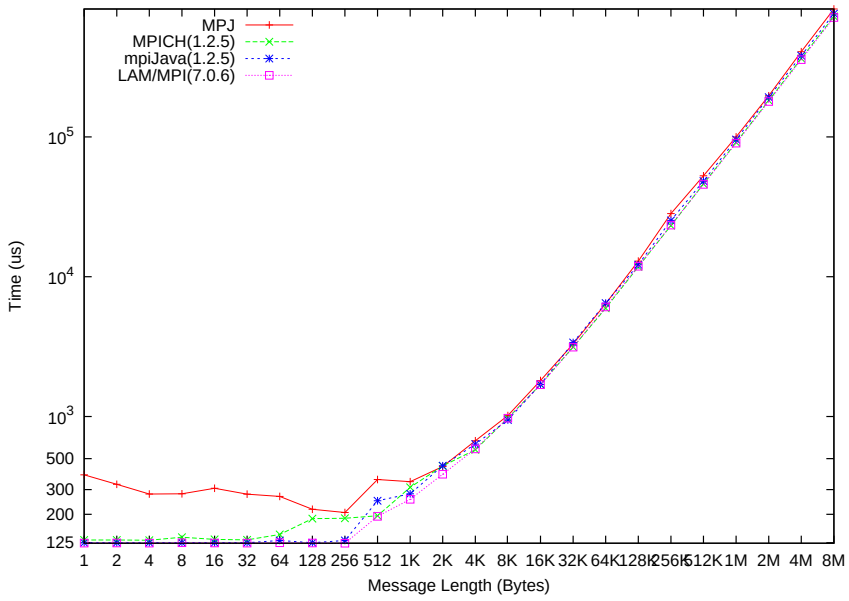

Fig. 2. A comparison of transfer time of MPJ with MPICH, LAM/MPI, and mpiJava

The runtime does not assume a shared file system, and uses dynamic class loading and the reflection API to load the necessary JAR (Java Archive) files and start the execution.

5 Preliminary performance evaluation

This section presents a ping-pong comparison of MPJ with other popular messaging libraries. The tests were conducted on two compute nodes of the DSG cluster "StarBug", each with a Dual Xeon (Prestonia) processor with clock speed of 2.8 GHz running Debian GNU/Linux (Kernel 2.4.30). These nodes were connected by Fast Ethernet.

Figure 2 shows plots of the transfer time comparison between MPJ, mpiJava [11], MPICH [5], and LAM/MPI [7]. We define latency as, "the time to to transfer one byte message". The latency of LAM, MPICH, mpiJava, and MPJ is 125, 125, 127, and 320 s respectively. The reason for higher latency of MPJ is that currently the sender writes the control message and the data in two separate writes. Also, the receiver first receives the control message followed by the actual data followed by the data receive operation.

Figure 3 indicates that LAM and MPICH almost achieve 90% of the available bandwidth, which is the theoretical maximum on Fast Ethernet. mpiJava acheives 84 Mbps; as a result of the JNI overhead. This overhead appears mainly because of an additional copying of the data from the JVM onto the native OS

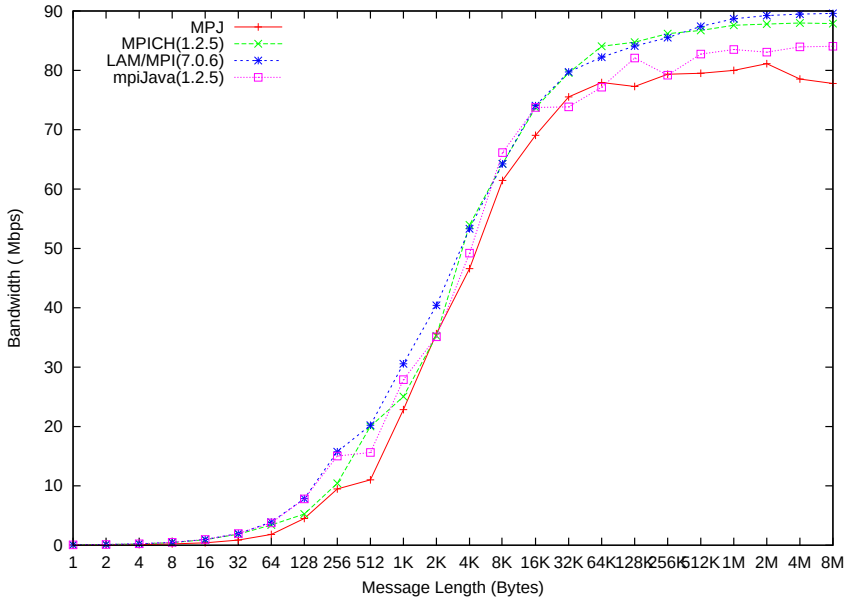

Fig. 3. A comparison of bandwidth of MPJ with MPICH, LAM/MPI, and mpiJava

buffer. This overhead becomes much more significant for large messages. More details about the JNI overhead can be found in [16]. Lastly, MPJ achieves almost 80 Mbps, which is 80% of the available bandwidth. The reason for 10% overhead is the creation time of the buffer for large messages. Whenever a `Send()/Recv()` is called at the MPJ level, it creates a new `mpjbuf.Buffer` object that holds the data. The creation cost of this buffer is almost 10% of the total transmission time for large messages. We plan to use "buffer pools" to avoid the overhead of creating a buffer for each `Send()/Recv()` method.

6 Conclusion

In this paper, we have presented MPJ, which is an implementation of MPI in pure Java. As part of MPJ, we have developed a Java NIO based communication device. Also, high and base level API has been implemented. Our design provides the capability of swapping in or out different devices, using a pluggable architecture. Such a design allows the applications to choose the communication protocol that best suits their needs. To demonstrate the flexibility of our design, we are also developing gmdev, smpdev, and a native MPI-2 device. With native MPI device, it will also be possible to access advanced MPI features of the native implementation. We believe that MPJ provides a tool that could be used to parallelise Java simulations.

References

1. Mark Baker, Hong Ong, and Aamir Shafi. A status report: Early experiences with the implementation of a message passing system using Java NIO. Technical report, DSG, October 2004. http://dsg.port.ac.uk/projects/mpj/docs/baker04.pdf.
2. Pedro Bizarro, Luís Moura Silva, and João Gabriel Silva. JWarp: A Java library for parallel discrete-event simulations. *Concurrency - Practice and Experience*, 10(11-13):999–1005, 1998.
3. Bryan Carpenter, Vladimir Getov, Glenn Judd, Tony Skjellum, and Geoffrey Fox. MPJ: MPI-like message passing for Java. *Concurrency: Practice and Experience*, 12(11), 2000.
4. Geoffrey C. Fox and Wojtek Furmanski. Java for parallel computing and as a general language for scientific and engineering simulation and modeling. *Concurrency: Practice and Experience*, 9(6):415–425, 1997.
5. William Gropp and Ewing Lusk. MPICH - a portable implementation of MPI. www.mcs.anl.gov/mpi/mpich.
6. The Java Service Wrapper project. http://sourceforge.net/projects/wrapper.
7. LAM/MPI Parallel Computing. www.lam-mpi.org.
8. Sang Boem Lim, Bryan Carpenter, Geoffrey Fox, and Han-Ku Lee. A device level communication library for the HPJava programming language. In *IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, November 2003.
9. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tenessee, Knoxville, TN, June 1995. www.mcs.anl.gov/mpi.
10. The MONARC project. www.cern.ch/MONARC.
11. The mpiJava project. www.hpjava.org/mpiJava.html.
12. The GM-2 message passing library. www.myri.com.
13. The Java New I/O Specifications. http://java.sun.com/j2se/1.4.2/docs/guide/nio.
14. N. T. Padial-Collins, W. B. VanderHeyden, D. Z. Zhang, E. D. Dendy, and D. Livescu. Parallel operation of CartaBlanca on shared and distributed memory computers. *Concurrency and Computation: Practice and Experience*, 16(1):61–77, 2004.
15. Y. M. Teo, Y. K. Ng, and B. S. S. Onggo. Conservative simulation using distributed-shared memory. In *PADS '02: Proceedings of the sixteenth workshop on Parallel and distributed simulation*, pages 3–10, Washington, DC, USA, 2002. IEEE Computer Society.
16. Matt Welsh and David Culler. Jaguar: enabling efficient communication and I/O in Java. *Concurrency: Practice and Experience*, 12(7):519–538, 2000.