MPJ: A New Look at MPI for Java

Mark Baker[1], Bryan Carpenter[2], Aamir Shafi[1]

1. Introduction

The Message Passing Interface (MPI) was introduced in June 1994 as a standard message passing API for parallel scientific computing. The original MPI standard had language bindings for Fortran, C and C++. A new generation of distributed, Internet-enabled computing inspired the later introduction of similar message passing APIs for Java [1][2]. Current implementations of MPI for Java usually follow one of three approaches: use JNI to invoke routines of the underlying native MPI that acts as the communication medium; implement message passing on top of Java RMI—remote method invocation of distributed objects; or implement high performance MP in terms of low-level "pure" Java communications based on sockets. The latter approach is preferred by some as it achieves good performance and ensures a truly portable system.

But experiences gained with these implementations suggest that there is no 'one size fits all' approach. 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. The challenging issue is how to manage these contradictory requirements.

The MPJ project described here is developing a next generation MPI for Java—building on the lessons learnt from earlier implementations, and incorporating a *pluggable architecture* that can meet the varying requirements of contemporary scientific computing. Our initial work focuses on improvements to the underlying messaging infrastructure (transport layer), though at the run-time level we must also address the security and fault-tolerance requirements of the Grid.

2. MPJ Design and Implementation

To address the outlined issues, we are implementing Message Passing in Java (MPJ). This follows a layered architecture based on an idea of device drivers. The idea is analogous to UNIX device drivers, and provides the capability to swap layers in or out as needed. MPJ implements the advanced features of the MPI specifications, which include derived-datatypes, virtual topologies, different modes of send, and collective communications. The high level features are implemented in Java, but if the underlying device uses a native MPI, it will also be possible to cut through directly to the native implementation.

Figure 1 provides a layered view of the messaging system showing the two device levels. The high and base level rely on the *MPJ device* and *xdev* level for actual communications. There are two implementations of the *mpjdev* level. The first uses JNI wrappers to a native MPI library, whereas the other sits on top of *xdev*. This is a newly proposed Java portability layer that sits between mpjdev (described in earlier works) and the physical hardware. Figure 1 also shows three implementations of xdev, shared memory device (*smpdev*), Java NIO device (*niodev*), and GM communications device (*gmdev*).

[1] Distributed Systems Group, University of Portsmouth
[2] Open Middleware Infrastructure Institute, University of Southampton

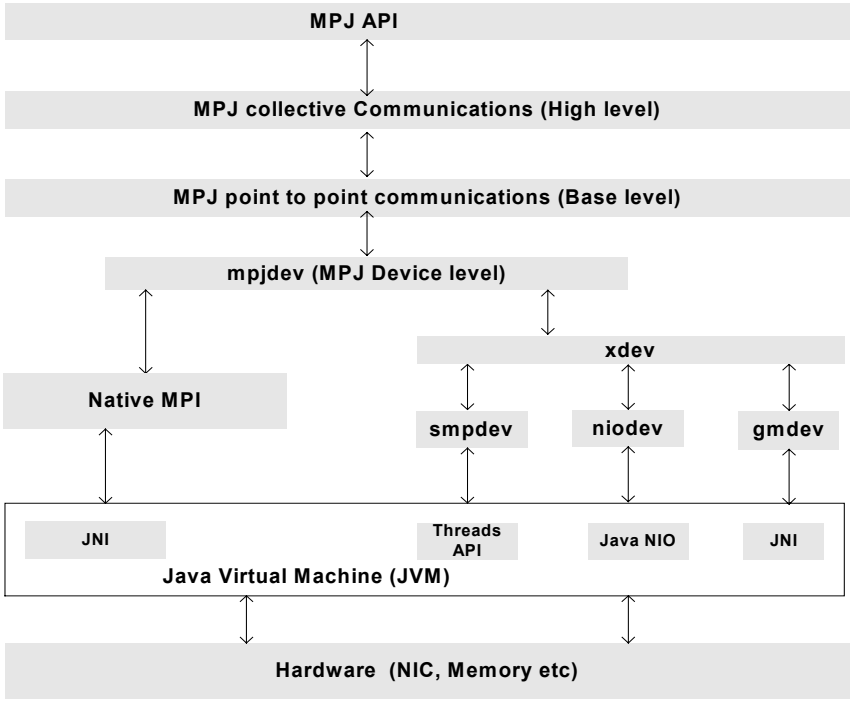

Figure 1: MPJ design

Previously, the task of bootstrapping MPI processes over a collection of machines was performed using RSH/SSH based scripts. More recently, runtime infrastructures like MPICH's SMPD (Super Multi Purpose Daemon) and LAM/MPI's runtime infrastructure provide an alternative solution. But these systems do not use Java: the portable nature of the Java language is one of the biggest advantages for implementing a messaging system, which will be compromised if MPJ relies on non-portable bootstrapping strategies. The new Java runtime will facilitate instantiation of MPJ programs from emerging Grid toolkits, like those based on WSRF and WS-I+ specifications.

3. Conclusion

Our initial performance evaluation shows MPJ achieves comparable performance to C MPI libraries. Additionally, it 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. Further details about implementation and performance evaluation will be included in the full paper.

References

[1] Bryan Carpenter, Vladimir Getov, Glenn Judd, Tony Skjellum, and Geoffrey Fox. MPJ: MPI-like Message Passing for Java. *Concurrency: Practice and Experience*, Volume 12, Number 11. September 2000.

[2] Mark Baker, Bryan Carpenter, Sung Hoon Ko, and Xinying Li. mpiJava: A Java interface to MPI. Presented at First UK Workshop on Java for High Performance Network Computing, Europar 1998.