

# MPJ: A New Look at MPI for Java

Mark Baker<sup>1</sup>, Bryan Carpenter<sup>2</sup>, Aamir Shafi<sup>1</sup>

## 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

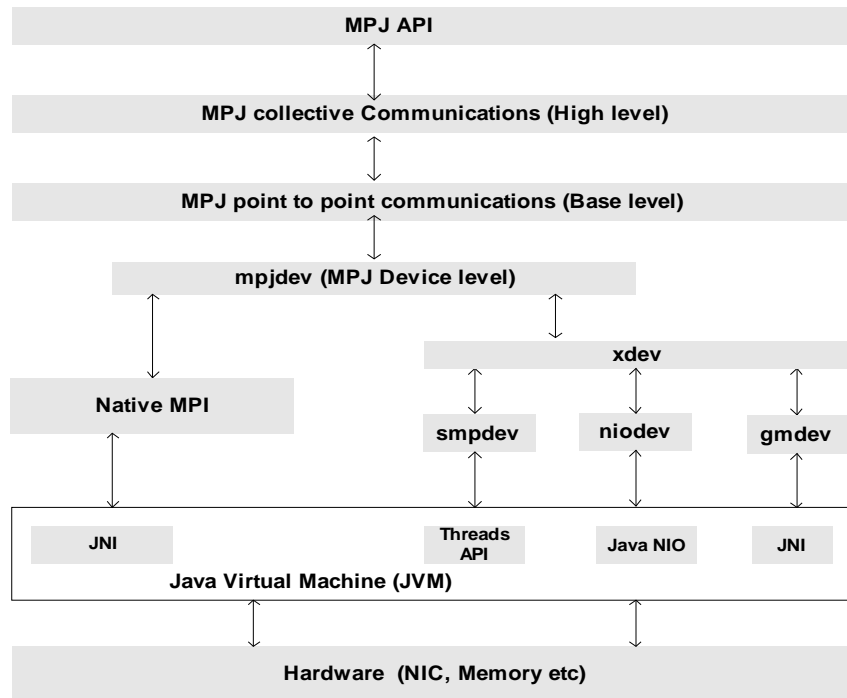
To address the outlined issues, we are implementing Message Passing in Java (MPJ). This follows a layered architecture based on an idea of *device drivers*. The idea is analogous to UNIX device drivers, and provides the capability to swap layers in or out as needed. MPJ implements the advanced features 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 directly access 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*).

---

<sup>1</sup> Distributed Systems Group, University of Portsmouth

<sup>2</sup> 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. 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, MPICH, and LAM/MPI. We define latency as, "the time to transfer one byte message". The latency of LAM, MPICH, mpiJava, and MPJ is 125, 125, 127, and 320  $\mu$ 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 achieves 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 buffer. This overhead becomes much more significant for large messages. 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.

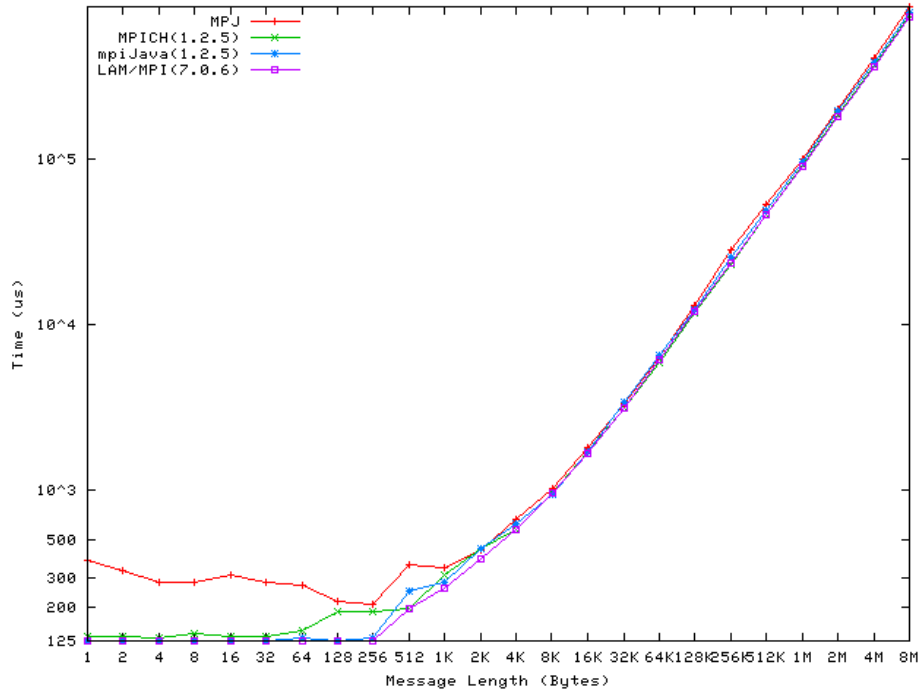


Figure 2: A comparison of transfer time of MPJ with MPICH, LAM/MPI, and mpiJava

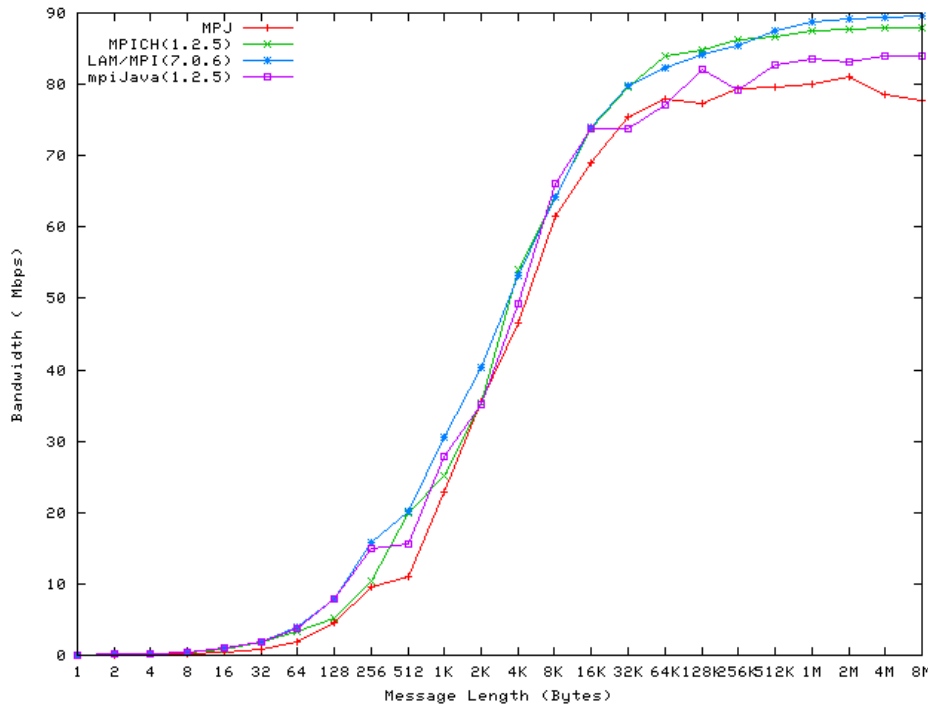


Figure 3: A comparison of the bandwidth of MPJ with MPICH, LAM/MPI, and mpiJava

## 4. Conclusions

In this short paper we have presented an overview of MPJ, which is an implementation of the MPI standard in pure Java. 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. The initial performance evaluation has revealed several optimisation areas for Java NIO based device.

## 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.