MPJ Express: Towards Thread Safe Java HPC

Mark Baker, Bryan Carpenter, and Aamir Shafi

Abstract

MPJ Express is a thread-safe Java messaging library that provides a full implementation of the *mpiJava 1.2* API specification. This specification defines a MPI-like bindings for the Java language. We have implemented two communication devices as part of our library, the first, called *niodev* is based on the Java New I/O package and the second, called *mxdev* is based on the Myrinet eXpress library. MPJ Express comes with an experimental runtime, which allows portable bootstrapping of Java Virtual Machines across a cluster or network of computers. In this paper we describe the implementation of MPJ Express. Also, we present a qualitative and quantitative comparison against various other C and Java messaging systems. A beta version of MPJ Express was released in September 2005.

I. Introduction

In this paper, we discuss MPJ Express (MPJE), a production quality implementation of MPI-like bindings for Java. A priority in implementing MPJ Express was to maintain compatibility with Java threads. The current trend towards SMP (Symmetric Multi-Processor) clusters underscores the importance of thread-safe HPC libraries. Using a thread-safe communication library to program such clusters is an alternative to traditional approaches like hybrid MPI and OpenMP code, or using shared memory devices in the MPI libraries. Java is one of the few mainstream programming languages that is multi-threaded by design, and this makes it an attractive language for programming SMP clusters, provided a thread-safe communication library is available. A focus of this paper is therefore the thread-safety of MPJ Express. Currently, we have implemented two thread-safe communication devices: *niodev* based on Java NIO (New I/O)[13], and *mxdev* based on the Myrinet eXpress(MX) library [12].

ACET, University of Reading
OMII, University of Southampton
DSG, University of Portsmouth

Soon after its release in 1996, Java became one of the "mainstream" programming languages of the software industry. It did not take long for researchers to suggest that it might also make a good option for high performance scientific computing. As a response to the appearance of several prototype MPI-like systems, the Message-Passing Working Group of the Java Grande Forum was formed in late 1998. This working group came up with an initial draft for a common messaging API, which was distributed at the Supercomputing conference in 1998. Since then, two APIs namely *mpiJava 1.2* [5] and *MPJ* [6] have been proposed. The main difference between these two APIs lies in the naming conventions of variables and functions.

There have been various implementations of Java messaging systems for HPC over the last decade. These systems typically follow one of three approaches: use JNI (Java Native Interface) [7] to interact with an underlying MPI [10] library; implement a Java messaging system from scratch using the likes of RMI [14]; or realize communications via a lower-level Java sockets API. Experience gained with these implementations suggests that there does not exist a universal approach that satisfies the conflicting requirements of end users. Using 100% pure Java ensures portability, but it might not provide the most efficient solution, especially in the presence of commodity high-performance hardware. It is important to address these contradictory requirements of portability and high performance in the design of Java messaging systems. To address this we have introduced a pluggable architecture [2], that adds a new device layer *xdev* below an already existing device layer *mpjdev* [9].

Section II presents related work. Section III recaps MPJE design, followed by a discussion on implementation in section IV. Section V evaluates the performance of MPJE. We conclude and discuss future work in Section VI.

II. Related Work

There have been various efforts over the last decade to develop a Java messaging system. Most of these projects were prototype implementations, thus their source-code is not available

and they are not maintained anymore. To our knowledge, only mpiJava, MPJ/Ibis, and MPJ Express are actively developed and maintained.

mpiJava [11] is a Java messaging system that uses JNI to interact with the underlying native MPI library. This project started in 1997 at NPAC in Syracuse University. mpiJava has been perhaps the most successful Java HPC messaging system, in terms of uptake by the community.

The mpiJava API is currently in version 1.2, although there are plans to move to the MPJ API in the future. The mpiJava library is not thread-safe, partly because it depends on an underlying native MPI library. Even if the native MPI library was thread-safe, some additional work would be needed to make the mpiJava wrappers thread-safe. mpiJava relies on native MPI's bootstrapping mechanism to start Java processes.

MPJ/Ibis [4] is an implementation of the MPJ API specification on top of Ibis [17]. The design philosophy of Ibis is similar to MPJ Express; it is possible to use 100% pure Java communication or use special HPC hardware like Myrinet. There are two pure Java devices in Ibis. The first called *TCPIbis* provides communication using the traditional `java.io` package. The second called *NIOIbis* uses the Java NIO package. Although TCPIbis and NIOIbis provide blocking and non-blocking communication at the device level, the higher-levels only use blocking versions of these methods. The communication devices used by MPJ/Ibis are not thread-safe. This library relies on scripts that use SSH for bootstrapping. Also, MPJ/Ibis does not fully implement the higher-level features of MPI like derived datatypes (apart from contiguous datatype), virtual topologies, and inter-communicators.

One fundamental difference between MPJ Express and MPJ/Ibis is *mpjbuf*, our buffering API [3]. In MPJ Express, the idea is to pack message onto a direct byte buffer and use these for communication. At the receiver side, the messages are unpacked from direct byte buffers onto user-defined primitive type arrays. This packing and unpacking poses some overhead for pure Java devices in MPJ Express, which is avoided in MPJ/Ibis pure Java devices. But this buffering API makes it possible to avoid JNI overheads, whereas the JNI devices for

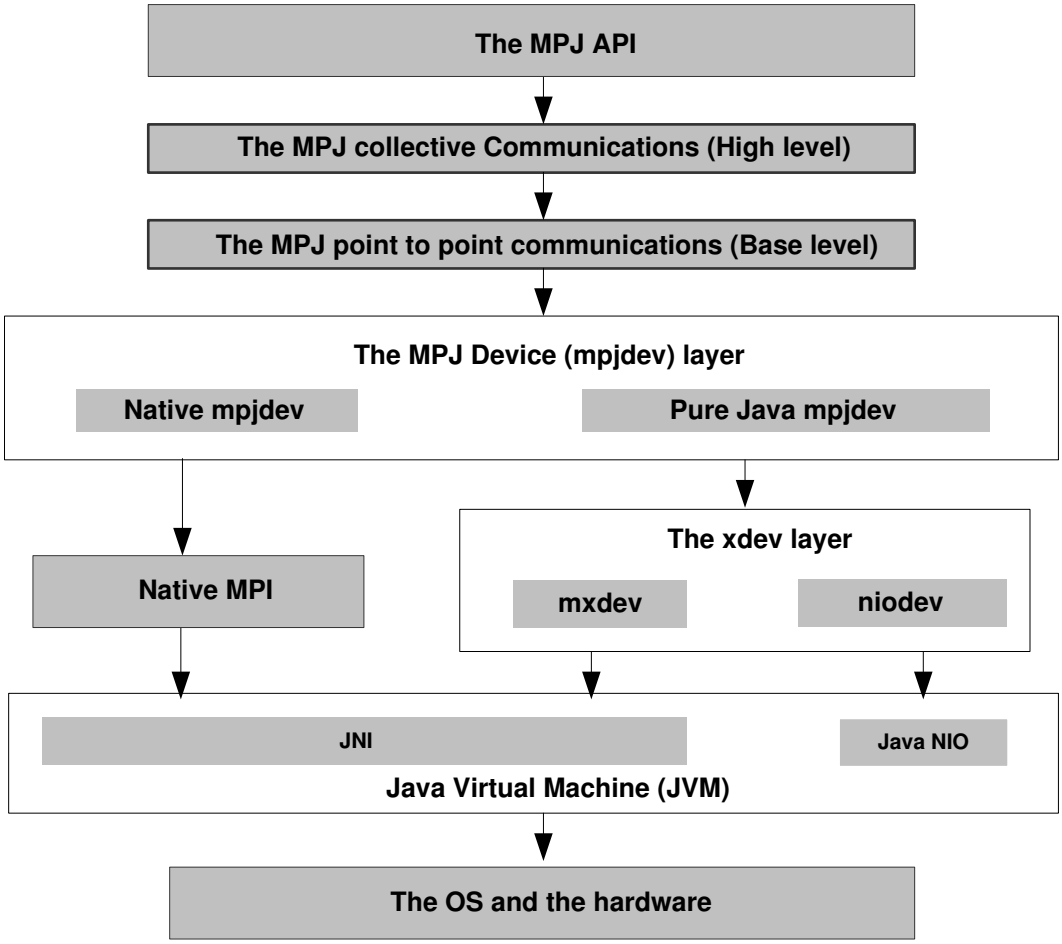

Fig. 1. MPJ Express Design

MPJ/Ibis cannot avoid data copying because they do not use direct byte buffers.

The presence of three (MPJE, mpiJava, and MPJ/Ibis) Java messaging systems shows a sustained interest in Java for HPC. All of these libraries are fairly stable and are publicly available along with their source-code.

III. MPJ Express Design

MPJE has a layered design that allows incremental development, and provides the capability to update and swap layers in or out as needed. Figure 1 is a layered view of the messaging system that shows MPJ Express levels: high-level, base-level, *mpjdev*, and *xdev*.

The high and base level rely on the *mpjdev* and *xdev* levels for actual communications, and interaction with the underlying networking hardware.

We envisage two implementations of the *mpjdev* level. The first implementation will provides JNI wrappers to native MPI implementations. It would also be possible to use higher-level features of native MPI library, instead of relying on the pure Java ones implemented at top levels of the design. The second implementation will use the lower level device called *xdev* to provide access to Java sockets or specialized communication libraries. *xdev* is not needed by the wrapper implementation because the native MPI is responsible for selecting and switching between the different communication protocols.

Figure 1 shows the two implementations of *xdev*. *mxdev* is the device for interconnect using the Myrinet eXpress (MX) library and *niodev* is based on Java NIO.

A. xdev *API*

The aim of this device layer is to provide the means to flexibly swap communication protocols. Also, we aim to keep the API simple and small, to minimize the overall development time of devices. The reason for introducing a new device layer is that *mpjdev* deals with ranks for MPI processes. This results in management of communicators and groups at *mpjdev* layer. Also, it is necessary to provide wrappers to native MPI's higher-level features. This essentially means that *xdev* does not deal with the higher level abstractions of MPI, like groups, communicators, and contexts. It only focuses on providing communication methods. We present the API of the *xdev* device layer in Figure 2.

The `peek()` method returns the most recently completed `Request` object. The idea is borrowed from the Myrinet eXpress library. This method can be used to implement `Waitany()` at the higher-level. We explain this in detail in Section IV-E.1.

IV. MPJ Express Implementation

In this section, we will discuss the implementation of the various MPI features in MPJ Express.

```
package xdev ;
public abstract class Device {
  public static Device newInstance(String dev) throws XDevException {
    .. initialize the appropriate implementation of xdev ..
  }
  public abstract ProcessID[] init(String[] args) throws XDevException;
  public abstract int getSendOverhead() ;
  public abstract int getRecvOverhead() ;
  public abstract ProcessID id();
  public abstract void finish() throws XDevException;
  public abstract mpjdev.Request isend(mpjbuf.Buffer buf, ProcessID destID,
                                int tag, int context) throws XDevException;
  public abstract void send(mpjbuf.Buffer buf, ProcessID destID,
                                int tag, int context) throws XDevException;
  public abstract mpjdev.Request issend(mpjbuf.Buffer buf, ProcessID destID,
                                int tag, int context) throws XDevException;
  public abstract void ssend(mpjbuf.Buffer buf, ProcessID destID,
                                int tag, int context) throws XDevException;
  public abstract mpjdev.Status recv(mpjbuf.Buffer buf, ProcessID srcID,
                                int tag, int context) throws XDevException;
  public abstract mpjdev.Status probe(ProcessID srcID, int tag,
                                        int context) throws XDevException;
  public abstract mpjdev.Status iprobe(ProcessID srcID, int tag,
                                        int context) throws XDevException;
  public abstract mpjdev.Request peek() throws XDevException;
}
```

Fig. 2. The *xdev* API

A. Point-to-Point Communication

The *send* and *receive* operations form the basic point to point communication mechanism. It is possible to use the blocking and non-blocking versions of these operations. In addition, the MPI specification document defines four modes of send method, namely normal, synchronous, ready, and buffered. In *niodev*, we have implemented these four modes using two communication protocols: *eager send* and *rendezvous* protocol. These protocols need not be implemented in the case of *mxdev* because the MX library has already implemented the communication protocols internally. We present the pseudocode for these communication protocols implemented for *niodev* in Section IV-A.1 and IV-A.2—they might serve as a blueprint for developing other thread-safe devices.

In *niodev*, each process connects to every other process with two NIO channels. The reason for two channels is that we use blocking mode for writing messages and non-blocking mode for reading messages, and the Java NIO API does not allow using both blocking and

```
isend:
  lock 'dest' channel
  send the data
  unlock 'dest' channel
  return a non-pending send request object
```

Fig. 3. Pseudocode for `isend` method (eager protocol)

non-blocking modes on a single channel. There is a separate lock (per destination) associated with each write channel, which means every thread that tries to write a message first acquires the associated lock. No lock is required for reading messages because only one thread receives messages. We call this thread the *input-handler* in the discussion below. It is often referred as a *progress engine* in related literature. This thread makes use of `select()` to process events associated with registered channels.

A.1 The *niodev* Eager Send Protocol

The eager send protocol is normally used for the communication of small messages, typically less than 128 Kbytes when using TCP/IP. 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 the actual data transmission. This minimizes the overhead of control messages that may dominate the total communication time for small messages.

Whenever a send method is called, the sender writes the message data into the channel assuming that the receiver will handle it as shown in the pseudocode in Figure 3. At the receiving side, there can be more than one scenario when the *input-handler* thread receives a message, depending on whether a matching receive method is already posted by the user or not. If a matching receive method has already been posted, the message is copied onto the memory specified by the user. If a matching receive has not yet been posted, then the message is stored in a temporary buffer, and copied to the user specified memory when subsequently the user calls the matching receive method. The pseudocode for the receiver can be seen in Figure 4 and Figure 5.

```
irecv:
  lock receive-communication-sets
  if irecv matches some receive request in pending-recv-request-set {
    copy data from input-buffer into user-buffer
    remove non-pending receive request from pending-recv-request-set
  }
  else {
    add pending receive request to pending-recv-request-set
  }
  unlock receive-communication-sets
  return the receive request
```

Fig. 4. Pseudocode for `irecv` method (eager protocol)

```
  loop {
    receive header
    lock receive-communication-sets
    if message matches a receive request in pending-recv-request-set {
      remove receive request from pending-recv-request-set
      unlock receive-communication-sets
      receive data into user-buffer
    }
    else {
      receive data into input-buffer
      add receive request to pending-recv-request-set
      unlock receive-communication-sets
    }
  }
```

Fig. 5. Pseudocode for input handler threads (eager protocol)

A.2 The *niodev* Rendezvous Protocol

The rendezvous protocol is used for communication of 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. For large enough messages, the overhead of this exchange of messages is negligible in terms of the overall communication cost.

When the sender intends to send a message a *ready-to-send* control message is sent. Figure 6 shows the psuedo-code for the user thread that sends such a control message. Note that there is a separate lock for *send-communication-sets* and destination channel. When a process calls a send method it acquires two locks—the first to manipulate *send-communication-sets* and the second to gain exclusive access to the destination channel. These locks are acquired one after the another, to avoid blocking other user threads sending messages to different destinations.

```
isend:
  lock send-communication-sets
  add send request to pending-send-request-set
  unlock send-communication-sets
  lock 'dest' channel
  send a ready-to-send message
  unlock 'dest' channel
  return pending send request
```

Fig. 6. Pseudocode for `isend` method (rendezvous protocol)

When the *input-handler* thread at the receiver end receives this message, it first checks *pending-recv-request-set* to see if a matching receive is posted or not. If it has been posted the *input-handler* thread sends back a *ready-to-recv* control message. The pseudocode for this can be seen in Figure 8. If it has not been posted the user-thread that posts the receive method is responsible for sending *ready-to-recv* control message as shown in Figure 7.

Back at the sender side, the input-handler thread receives a *ready-to-recv* message and forks a write thread that sends the actual data to the receiver as shown in Figure 8. The reason for forking a new thread is to avoid blocking the input-thread. Such blockage of input-thread could result in a deadlock if two processes are simultaneously sending large messages to each other. We opted to use blocking channels for sending data, because non-blocking ones would add unnecessary complexity to the device.

The *input-thread* receives the message data in a non-blocking fashion. This ensures that *input-thread* is never blocked and makes progress even if multiple messages are received simultaneously. If a full message is not received, then the corresponding receive request is added to `java.nio.channels.SelectionKey`, which is a token for representing the registration of a channel with a selector (*input-thread*). When the rest of the message is received, the pending receive request can be obtained to resume receiving of the data. The pseudocode for the *input-thread* is shown in Figure 8.

```
irecv:
  lock receive-communication-sets
  if irecv matches a receive request in pending-recv-request-set {
    unlock receive-communication-sets
    lock 'src' channel
    send 'ready-to-recv' message
    unlock 'src' channel
  }
  else {
    add pending receive request to pending-recv-request-set
    unlock receive-communication-sets
  }
  return the receive request
```

Fig. 7. Pseudocode for `irecv` method (rendezvous protocol)

A.3 *mxdev* Implementation

Our communication device *mxdev* uses JNI to communicate with the MX library. It does not implement any communication protocols because these protocols have been internally implemented by the MX library. An added advantage is that the communication functions provided by MX are thread-safe.

Because our buffering API *mpjbuf* makes it possible to use direct byte buffers, we have been able to avoid one of the main overhead of using JNI—copying data between the JVM (Java Virtual Machine) and the OS.

The MX library can be initialized with the `mx_init()` method. Once this is done, the `mx_open_endpoint()` method is used to listen for incoming connections. Each process opens up one endpoint, and all others connect to this endpoint using the `mx_connect()` method. The MX library provides non-blocking versions of standard and synchronous mode of the send operation. Also, it provides a non-blocking version of the receive operation. The signature of `mx_isend()` is:

```
mx_return_t mx_isend(mx_endpoint_t endpoint, mx_segment_t *segments_list,
             uint32_t segments_count, mx_endpoint_addr_t destination,
             uint64_t match_send, void *context, mx_request_t *request);
```

The second argument to `mx_isend()` is a pointer `segments_list`, followed by a third argument `segments_count`. This means that the data sent can belong to two or more

```
input handler thread:
  loop {
    if a channel is attached to selection key (rest of the data) {
      detach 'src' channel from selection key
      receive the data
      if full data has not been received {
        attach 'src' channel to selection key
      }
    }
    else {
      receive header
      if message is a ready-to-send message {
        lock receive-communication-sets
        if header matches some receive request in pending-recv-request-set {
          unlock receive-communication-sets
          lock 'src' channel
          send 'ready-to-recv' message to sender
          unlock 'src' channel
        }
        else {
          add pending receive request to pending-recv-request-set
          unlock recv-communication-sets
        }
      }
      if message is a ready-to-receive message {
        fork a rendez-write-thread
          rendez-write-thread:
            lock send-communication-sets
            remove a non-pending send request from pending-send-request-set
            unlock send-communication-sets
            lock 'dest' channel
            send the data
            unlock 'dest' channel
      }
      if message is data {
        lock receive-communication-sets
        remove the matching receive request from pending-recv-request-set
        unlock receive-communication-sets
        receive the data
        if full data has not been received {
          attach 'src' channel to selection key
        }
      }
    }
  }
```

Fig. 8. Pseudocode for input handler threads (rendezvous protocol)

contiguous sections in memory. Our buffering API [3] stores primitive datatypes in a so-called *static* section of the buffer, whereas the object datatypes are stored in a so-called *dynamic* section. This feature of sending data from multiple contiguous segments makes it possible to send static and dynamic sections in one `mx_isend()` call.

B. Thread Safety

The MPI 2.0 specification introduced the notion of *thread compliant* MPI implementation. This document introduced four levels of thread-safety. `MPI_THREAD_SINGLE` means only one thread executes in a MPI process. `MPI_THREAD_FUNNELED` means that the MPI process maybe multi-threaded, but only the main thread could communicate to other processes. `MPI_THREAD_SERIALIZED` means that multiple threads can make MPI calls but only one at a time. `MPI_THREAD_MULTIPLE` means that multiple threads can do communication without any restriction. The newly provided methods allow initializing the thread environment with a requested level of thread-safety and querying for the level of thread-safety that a MPI implementation provides. There are no Java bindings for the MPI 2.0 specification document—we plan to produce Java bindings for thread-safety related variables and methods.

MPJ Express runs with level `MPI_THREAD_MULTIPLE` by default. A MPJE process can have multiple threads, which can communicate with other processes without any restriction. We believe MPJ Express is the first Java HPC messaging library to support this level of thread-safety. Only a handful of C MPI libraries support `MPI_THREAD_MULTIPLE`.

To verify our claim, we wrote multi-threaded test cases that are part of MPJ Express test suite. These test cases start multiple threads for a single MPJE process. These threads communicate with other process. When the message is received at the receiver, the contents of the message are verified. Another verification test is called `ProgressionTest`. In this test, one of the thread running in a multi-threaded MPJE process blocks itself and we check if this halts the execution of other threads in the same process. This test revealed that our approach successfully meets this criterion of multi-threaded libraries too.

C. Derived Datatypes

Derived datatypes were introduced in the MPI specifications to allow communication of heterogeneous and non-contiguous data. It is possible to achieve some of the same goals by communicating Java objects, but there are concerns about the cost of object serialization—MPJ Express relies on JDK's default serialization.

There are four types of derived datatypes; contiguous, indexed, vector, and struct. Derived datatypes are important as they might simplify application development. For instance, the vector datatype consists of elements of the same type at non-contiguous locations. Imagine a 4x4 matrix stored in a float array. It is possible to send first column of this matrix using the vector datatype, by specifying a `blockLength` of 1 and `stride` of 4 when initializing the datatype. When the send method is called, the first column is copied to a contiguous area, which is used for the actual send. This is made possible in MPJ Express by our buffering API *mpjbuf*. More details about *mpjbuf* can be found in [3].

D. The Runtime System

An important component of a Java messaging system is the mechanism used for bootstrapping MPJE processes across various platforms. A challenge here is to make the mechanism cope with heterogeneous platforms. If the compute-nodes are running a UNIX-based OS, it is possible to remotely execute commands using SSH. But if the compute-nodes are running Microsoft Windows, these utilities are not universally available. The MPJE runtime provides a unified way of starting MPJE processes on compute-nodes irrespective of the operating system. The runtime system consists of two modules. The daemon module executes on compute-nodes and listens for requests to start MPJE processes. The daemon is a Java application listening on an IP port, which starts a new JVM whenever there is a request to execute an MPJE process. MPJ Express uses the Java Service Wrapper Project [8] software to install daemons as a native OS service. The `mpjrun` module acts as a client to the daemon module. This module may be invoked on, for example, a cluster's head-node.

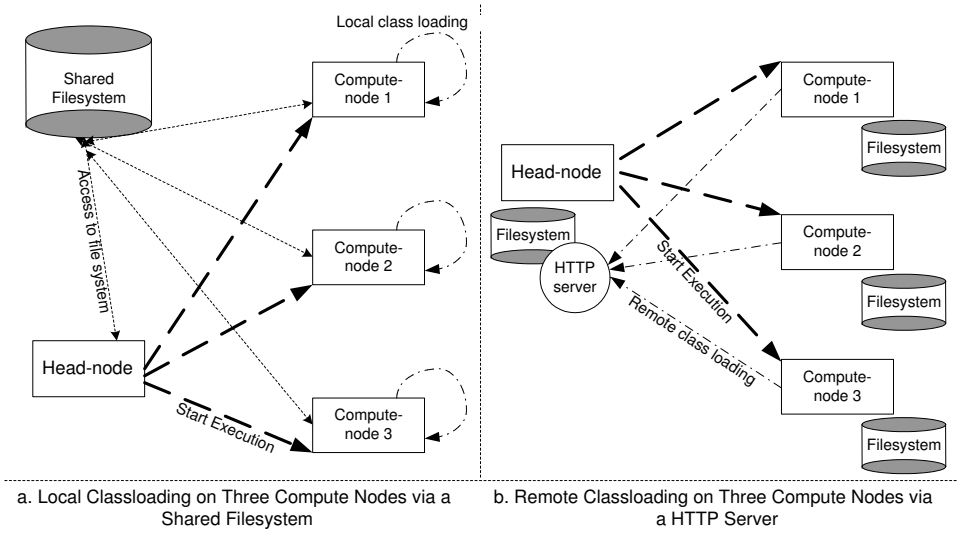

Fig. 9. The Runtime Infrastructure

It will contact daemons, which will start MPJE processes in a new JVM.

Java allows one to run applications using class files in an open directory structure, or bundled as a JAR file. The MPJE runtime allows the execution of MPJE applications in either format. Users may want to load MPJE JARs and classes either remotely or locally on the compute-nodes as shown in Figure 9. With the remote loader, it is possible to load all classes (application and MPJE code) from the users development node as shown in Figure 9b. This is useful in scenarios when there is no shared file system and the code is constantly being modified at the head-node. With the local loader, it is possible to load all classes (application and MPJ Express code) from the compute-node as shown in Figure 9a. This might be useful if there is a shared file system. As all classes are loaded locally, this might provide better performance in comparison to remote loader.

E. Qualitative Analysis of MPJ Express

In this section, we discuss some features of MPJ Express emphasizing that we have made an effort to ensure a production quality implementation. Due to space restrictions, we limit our discussion to implementation of `Waitany()` method and `Recv()` using wildcard `MPI.ANY_SOURCE`.

E.1 Multi-Threaded Implementation of `Waitany()`

A straightforward way of implementing `Waitany(Request[] requests)` method is to poll over a set of requests passed as an argument. This approach is not efficient in a multi-threaded setting because this can cause CPU starvation for any computation that might be running in parallel.

In MPJ Express, the actual implementation of `Waitany()` is at the *mpjdev* level. At this level, we use the `peek()` method provided by the *xdev* API. A `WaitAny` object has a reference to all `Request` objects, which were passed as argument to `Waitany()`. Also, each `Request` object stores a reference to `WaitAny` object, if a `Waitany()` method has been called for this `Request` object. Otherwise, the `WaitAny` object reference in `Request` object is `null`.

The `Waitany()` method accepts an array of active `Request` objects. We call `Test()` method for each element of `Request` objects array to check if any of them has completed. If one of them has completed, then `Waitany()` would return immediately. Otherwise, a `WaitAny` object is initialized and added to a static `WaitanyQue` object. In MPJ Express, it is possible that multiple threads might be calling `Waitany()` at the same time. Thus, we need to work out which one would call the `peek()` method.

Each thread that might be calling `Waitany()` has a unique reference to `WaitAny` object, that has been added to `WaitanyQue` object. The `WaitAny` object at the front of this queue is now responsible for calling the `peek()` method. Note that the `peek()` is a blocking method, that returns the most recently completed `Request` object. All the other `WaitAny` objects in different threads call `WaitAny.waitfor()` method.

When the `peek()` method called by the front `WaitAny` object returns a reference to most recently completed `Request` object, three scenarios are possible. The first scenario is that the returned `Request` object is associated with the calling `WaitAny` object. In this case, we wakeup the next `WaitAny` object in `WaitanyQue` object that is now responsible for calling the `peek()` method. The second scenario is that the returned `Request` object is associated

with a `WaitAny` object in `WaitanyQue` object. In this case, it is removed from the queue and the `Waitany.wake()` is called for it. The third and the last scenario is that no `Waitany()` method has been called for the returned `Request` object. If the returned `Request` object's `WaitAny` object reference is `null`, then we ignore this `Request` object.

E.2 Implementation of Receive method for `MPI.ANY_SOURCE`

In MPJ Express, we provide an efficient implementation for receiving messages with `MPI.ANY_SOURCE`. Whenever a receive method is called and the message has not already arrived, then a receive request is added to *pending-recv-request-set* as shown in Figure 7.

A message can be identified uniquely using `context`, `tag`, and `src`. Each receive request generates four keys because it is possible to use wildcards for `src` and `tag`. The first key is `(context, tag, src)`, the second is `(context, ANY_TAG, src)`, the third is `(context, tag, ANY_SOURCE)`, and the fourth is `(context, ANY_TAG, ANY_SOURCE)`. The incoming messages are matched using these keys to find the appropriate `request` object.

V. Performance Evaluation

In this section, we evaluate the performance of MPJ Express. Section V-A presents a qualitative comparison, followed by the conventional transfer time and throughput comparisons in Sections V-B, V-C, and V-D on Fast Ethernet, Gigabit Ethernet, and Myrinet, respectively.

We used MPJ Express (version 0.23), MPICH (version 1.2.5.2), MPJ/Ibis (version 1.2.1), LAM/MPI (version 7.0.6), and mpiJava (version 1.2.5). We used two devices for MPJ/Ibis: *TCPIbis* and *NIOIbis* are based on `java.io` and `java.nio` packages respectively. To differentiate between the two, we have named them MPJ/Ibis(TCPIbis) and MPJ/Ibis(NIOIbis). We also added *mpjdev* to our comparison to better understand the performance of MPJ Express. MPJ Express uses *mpjdev*, which in turn relies on *niodev* on Fast and Gigabit Ethernet and *mxdev* on Myrinet.

The test environment for collecting the performance results was a cluster at the DSG

called *StarBug*, consisting of 8 dual Intel Xeon 2.8 GHz PCs using the Intel E7501 chipset. The PCs were equipped with 2 Gigabytes of ECC RAM with 533 MHz Front Side Bus (FSB). The motherboard (SuperMicro X5DPR-iG2) is equipped with 2 onboard Intel Gigabit LAN adaptors with one 64-bit 133 MHz PCI-X slot and one 64-bit 66 MHz PCI slot. The PCs were connected together through a 24-port Ethernet switch. The PCs were running the Debian GNU/Linux with the 2.4.32 Linux kernel. The software used for the Intel Gigabit adaptor was the proprietary Intel e-1000 device driver. The JDK version that has been used for these tests for mpiJava and MPJE is Sun JDK 1.5 (Update 6). The C compiler used was GNU GCC 3.3.5.

The transfer time and latency is calculated using a modified ping-pong benchmark. While using conventional ping-pong benchmarks, we noticed variability in timing measurements. The reason is that the network card drivers used on our cluster have 64 microseconds network latency. The network latency of the card drivers is an attribute that determines the polling interval for checking new messages. In our modified technique, we introduced random delays before the receiver sends the message back to the sender. Using this approach, we were able to negate the affect of network card latency. We omit the details of this technique here because of space limitations, and plan to present it in a separate publication.

A. Qualitative Analysis

We presented our technique of implementing `Recv()` method using `MPI.ANY_SOURCE` in Section IV-E.2. To quantify the performance boost of this approach against other implementations, we wrote a test case with two processes. Each process calls non blocking receive with MPI.ANY_SOURCE for hundred messages at the start, does multiplication of two square matrix (3000x3000). At the end of this computation, each process sends hundred messages to the other process. We ran this test with MPJ Express and MPJ/Ibis for comparison. We found out that matrix multiplication at process 0 was 11% faster when using MPJ Express.

B. Transfer Time and Throughput Comparison on Fast Ethernet

Figure 10 and 11 shows transfer time and throughput comparison of MPJ Express against other messaging systems.

The latency (transfer time of one byte) of the C MPI library is the lowest of all. This is understandable because of lower latency of the C ping-pong benchmarks than their Java counterparts. mpiJava follows C MPI library because essentially it is using the same messaging mechanism. MPJ/Ibis and MPJ Express use pure Java, which is the main cause of slightly higher latency. The latency of MPJ Express is 164 microseconds, which is higher than MPJ/Ibis (144 microseconds for TCPIbis and 143 microseconds for NIOIbis). The latency of *mpjdev* is slighter lower than MPJ Express.

The throughput achieved at 16 Mbyte message size is more than 84% of the maximum for all systems. mpiJava achieves 84% and the difference to other messaging systems could be attributed to additional data copying incurred by JNI. LAM/MPI, MPJ/Ibis (using TCPIbis and NIOIbis) achieve 90%, followed by MPICH and MPJ Express. The drop at 128 Kbytes message size for MPICH, mpiJava, and MPJ Express is due to change of communication protocol from eager send to rendezvous.

C. Transfer Time and Throughput Comparison on Gigabit Ethernet

Most messaging libraries are not fine tuned to run on Gigabit Ethernet. This means that understanding the performance of these libraries can be non-trivial. There are many factors that could play a vital part in the overall performance on Gigabit Ethernet. Out of these, socket send and receive buffer size is probably the most influential. To have a fair comparison, we changed the default socket buffer size (send and receive) to 512 Kbytes for all messaging libraries.

Figure 12 and 13 show transfer time and throughput comparison on Gigabit Ethernet. The behavior is similar to Fast Ethernet—the latency values have been reduced due to a faster network technology. The throughput comparison in Figure 13 shows that LAM/MPI,

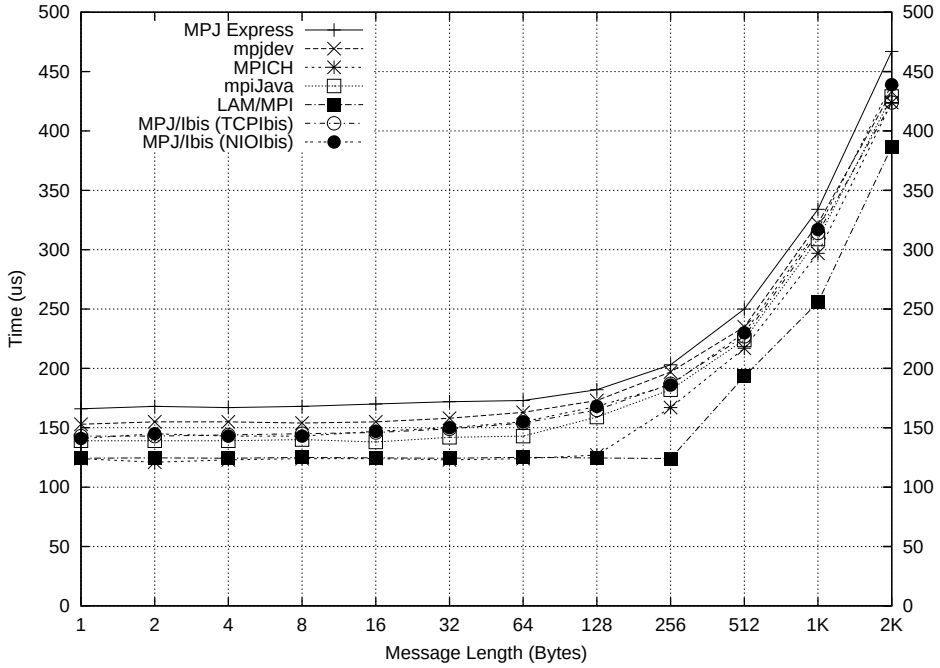

Fig. 10. Transfer Time Comparison on Fast Ethernet

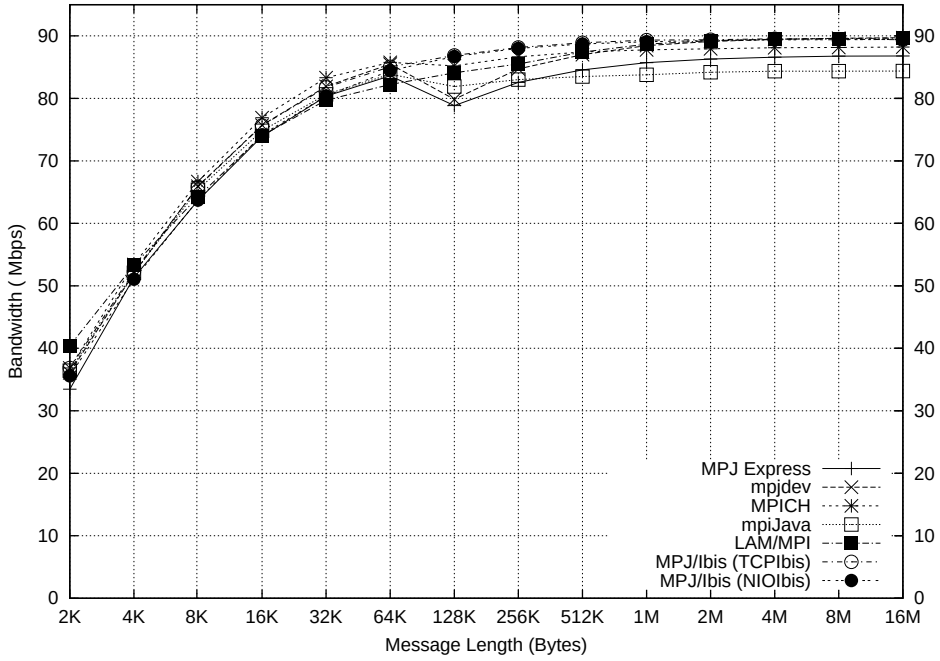

Fig. 11. Throughput Comparison on Fast Ethernet

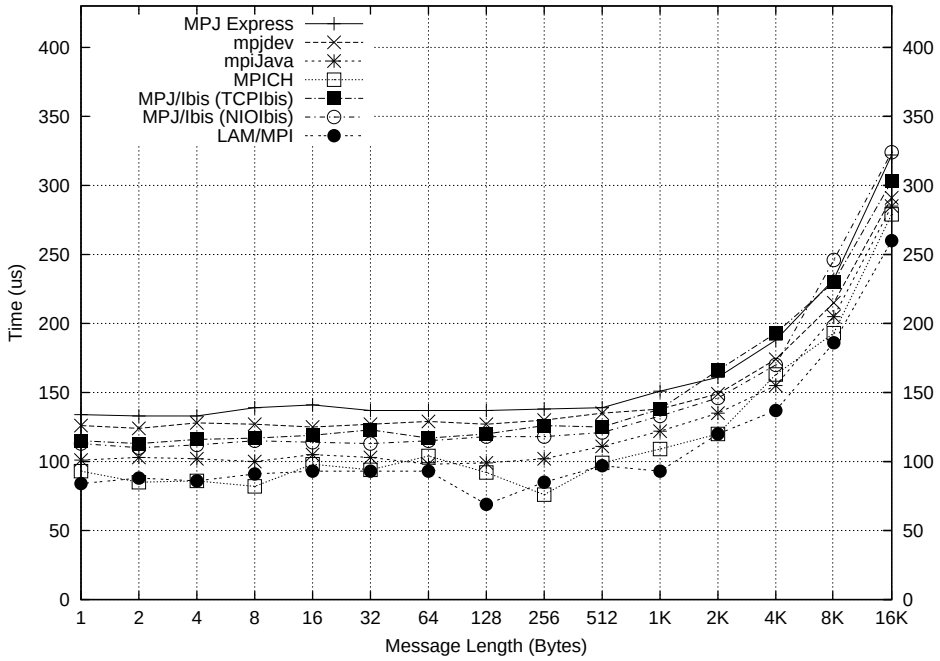

Fig. 12. Transfer Time Comparison on Gigabit Ethernet

MPJ/Ibis (TCPIbis), and MPJ/Ibis (NIOIbis) achieve 90% of total bandwidth. MPICH, MPJ Express, and mpiJava lag behind achieving 76%, 68%, and 60% throughput respectively. Although *mpjdev* achieves 90% of bandwidth for 16 Mbyte message, MPJ Express manages to reach 68%.

D. Transfer Time and Throughput Comparison On Myrinet

These tests were conducted on the same cluster, except we used 2 Gigabit Myrinet. The version of Myrinet eXpress library that we used was 1.1.0. Also, we used MPICH-MX (version 1.2.6.0.94) that runs on top of MX library.

Figure 14 and 15 show the transfer time and throughput comparison. The latency of MPICH-MX is 4 microseconds. MPJ Express and mpiJava have latency of 23 microseconds and 12 microseconds respectively. Throughput achieved by MPICH-MX is 1800 Mbps for 16 Mbytes. It is followed by MPJ Express that achieves 1097 Mbps for the same message size. mpiJava achieves a maximum of 1347 Mbps for 64 Kbytes messages. After this, there is a drop, bringing throughput down to 868 Mbps for 16 Mbytes message. *mpjdev* achieves

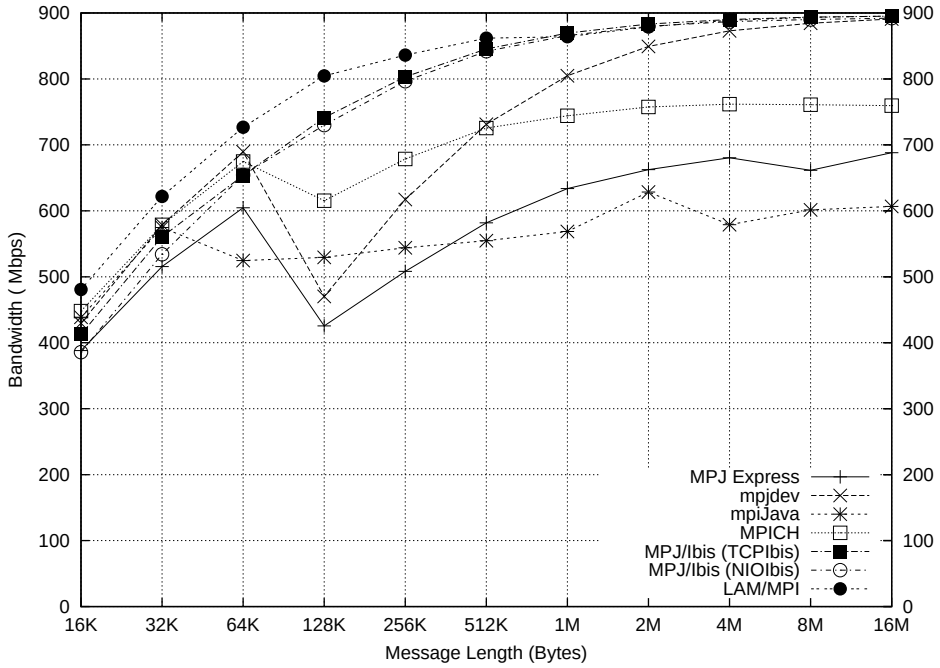

Fig. 13. Throughput Comparison on Gigabit Ethernet

1826 Mbps for 16 Mbyte message, which is more than what MPICH-MX achieves.

MPJ/Ibis provides a device *net.gm* that could be used, but we were unable to use it on our cluster. We have contacted the maintainers of MPJ/Ibis who are currently fixing the problem. For this draft, we use latency and throughput of *net.gm* reported in [1], which is 42 microseconds and 1100 Mbps, respectively. Note that these performance results are for *net.gm* device and not MPJ/Ibis, that would add some overhead.

E. Analysis of Transfer Time and Throughput Graphs

The main overhead in MPJ Express is the additional packing and unpacking required at the sender and the receiver, respectively. We copy the user message onto a direct byte buffer so that we can use NIO for pure Java device and avoid the JNI copying overhead for proprietary hardware devices. This could be verified by noting the difference between *mpjdev* and MPJ Express—there is no copying at *mpjdev* level. The reason for using direct byte buffers is that in the NIO package, the send and receive operations are provided by `SocketChannel`'s `write()` and `read()` methods. These methods only accept `ByteBuffer`

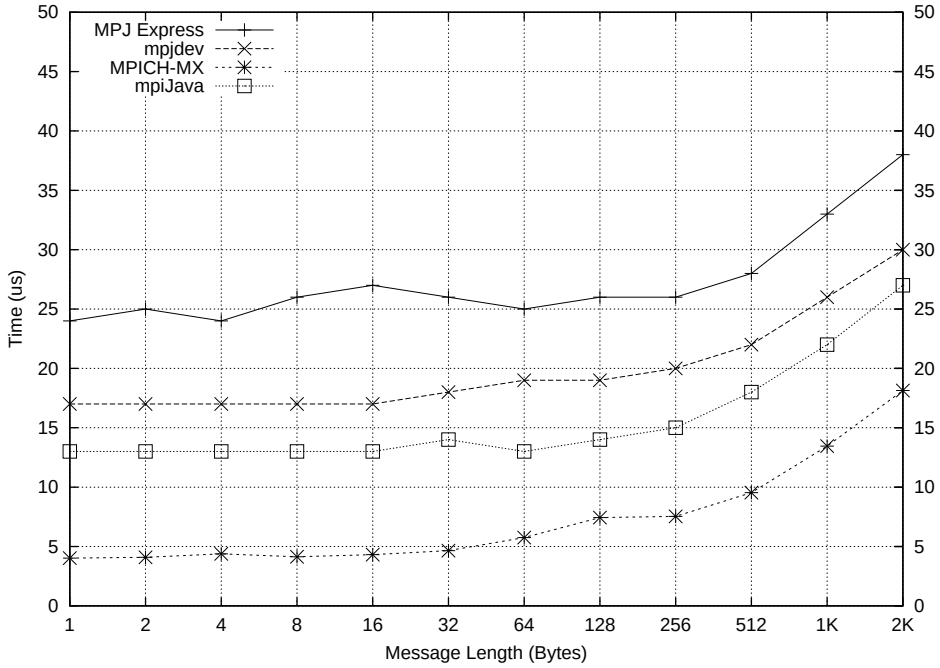

Fig. 14. Transfer Time Comparison on Myrinet

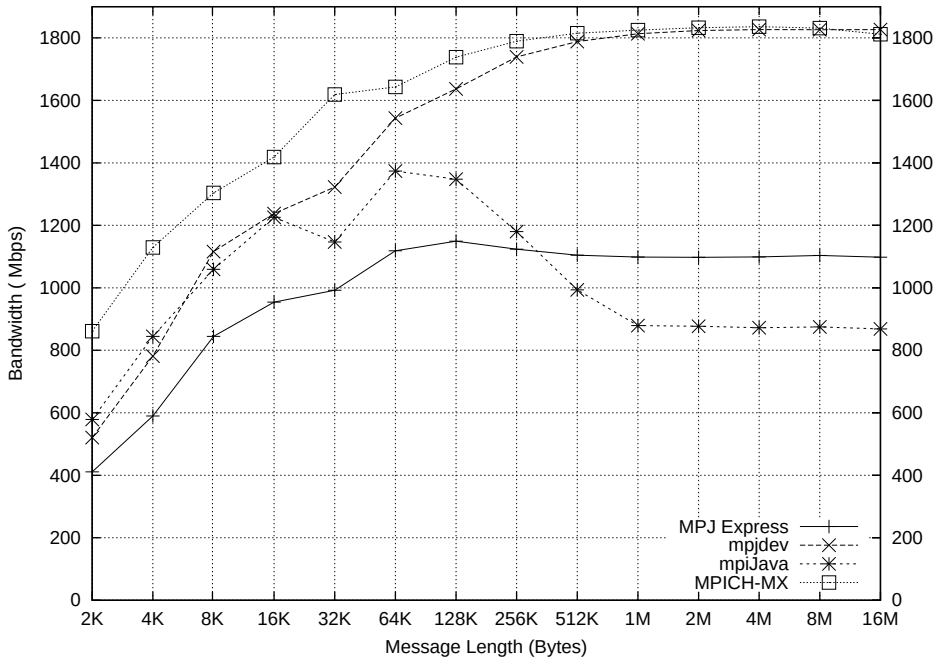

Fig. 15. Throughput Comparison on Myrinet

as an argument. Another advantage of using direct byte buffer is that it is possible to avoid the JNI copying overhead. The reason is that these direct byte buffers reside outside JVM heap and are not copied from heap onto native OS memory. This is evident by looking at throughput achieved by *mpjdev* on Myrinet—it achieves more throughput than MPICH-MX.

It is possible to avoid copying overhead in pure Java devices as shown by the performance of MPJ/Ibis on Fast and Gigabit Ethernet. But, this can be achieved using `java.io` package or using NIO in traditional `java.io` way. Using this approach, the message is directly written to and read from socket's input and output stream, respectively. The disadvantage is that the implementation cannot avoid JNI data copying and cannot use NIO's `select()` functionality. We believe that using NIO might provide better scalability on large clusters.

VI. Conclusions and Future Work

In this paper, we have presented MPJ Express, a thread-safe implementation of a MPI-like bindings for Java. MPJE implements all the high-level MPI features. Also, we have implemented two communication devices. The first device, *niodev*, is based on the Java NIO package, and provides portability by using 100% pure Java. We believe that using NIO's `select()` functionality can scale our device to large clusters. For example, we found out that it is possible to post any number of non-blocking receive methods using MPJ Express. Whereas, MPJ/Ibis, for example, fails with `cannot create native threads` exception while posting 650 simultaneous receive operations. The reason is that MPJ/Ibis starts a new thread for each send or receive operation. Our other communication device *mxdev* is based on the MX library and uses JNI to provide efficient communication.

We recently ported Gadget-2 [15] to Java using MPJ Express. Gadget-2 is a massively parallel structure formation code developed at the Max Planck Institute of Astrophysics. This code has been used in the "Millennium Simulation" [16]—the largest ever model of the Universe. The initial performance evaluation of the Java version revealed that it can achieve upto 70% of the original C Gadget-2's performance.

MPJ Express also comes with an experimental runtime that allows bootstrapping of Java processes. MPJ/Ibis and mpiJava do not have a runtime, which means that they rely on non-portable scripts that use SSH for bootstrapping. It could be quite cumbersome using these on Windows cluster or network.

We have benchmarked our system against various other messaging libraries and shown that MPJ Express is able to achieve comparable performance to other systems. There is an overhead associated with MPJ Express pure Java devices that can potentially be resolved by extending the *MPJ* API to allow communicating data to and from `ByteBuffer`s.

A beta version of MPJ Express was released in September, 2005 with *niodev* as the only option for communication. Since then, we have received valuable feedback from users and plan to provide a full stable release that includes *mxdev* in mid 2006.

References

[1] Olivier Aumage, Rutger Hofman, and Henri Bal. Netibis: An efficient and dynamic communication system for heterogeneous grids. In *Proc. of the Cluster Computing and Grid 2005 Conference (CCGrid 2005)*, Cardiff, UK, 2005. ACM/IEEE.

[2] Mark Baker, Bryan Carpenter, and Aamir Shafi. A Pluggable Architecture for High-Performance Java Messaging. *IEEE Distributed Systems Online*, 6(10), 2005.

[3] Mark Baker, Bryan Carpenter, and Aamir Shafi. An Approach to Buffer Management in Java HPC Messaging. In V. Alexandrov, D. van Albada, P. Sloot, and J. Dongarra, editors, *International Conference on Computational Science (ICCS 2006)*, LNCS. Springer, 2006.

[4] Markus Bornemann, Rob van Nieuwpoort, and Thilo Kielmann. MPJ/Ibis: A Flexible and Efficient Message Passing Platform for Java. In *PVM/MPI*, pages 217–224, 2005.

[5] Bryan Carpenter, Geoffery Fox, Sung-Hoon Ko, and Sang Lim. mpiJava 1.2: API Specification. http://citeseer.ist.psu.edu/ko99mpijava.html.

[6] Bryan Carpenter, Vladimir Getov, Glenn Judd, Anthony Skjellum, and Geoffrey Fox.

MPJ: MPI-like message passing for java. *Concurrency: Practice and Experience*, 12(11):1019–1038, 2000.

[7] Rob Gordon. *Essential JNI: Java Native Interface*. Prentice Hall PTR, Upper Saddle River, NJ 07458, 1998.

[8] The Java Service Wrapper project. http://sourceforge.net/projects/wrapper.

[9] Sang 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.

[10] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tenessee, Knoxville, TN, June 1995. www.mcs.anl.gov/mpi.

[11] The mpiJava project. www.hpjava.org/mpiJava.html.

[12] Myricom, The MX (Myrinet eXpress) library. http://www.myri.com.

[13] The Java New I/O Specifications. http://java.sun.com/j2se/1.4.2/docs/guide/nio.

[14] The Java Remote Method Invocation Specifications. http://java.sun.com/j2se/1.4.2/docs/guide/rmi.

[15] Volker Springel. The cosmological simulation code GADGET-2. *MON.NOT.ROY.ASTRON.SOC.*, 364:1105, 2005.

[16] Volker Springel, Simon D. M. White, Adrian Jenkins, Carlos S. Frenk, Naoki Yoshida, Liang Gao, Julio Navarro, Robert Thacker, Darren Croton, John Helly, John A. Peacock, Shaun Cole, Peter Thomas, Hugh Couchman, August Evrard, Joerg Colberg, and Frazer Pearce. Simulating the joint evolution of quasars, galaxies and their large-scale distribution. *Nature*, 435:629, 2005.

[17] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Rutger Hofman, Ceriel Jacobs, Thilo Kielmann, and Henri E. Bal. Ibis: a flexible and efficient Java based grid programming environment. *Concurrency and Computation: Practice and Experience*, 17(7-8):1079–1107, June 2005.