Procedia Computer Science

Volume 29, 2014, Pages 184–197

ICCS 2014. 14th International Conference on Computational Science

Design and Implementation of Hybrid and Native Communication Devices for Java HPC

Bibrak Qamar[1]*, Ansar Javed[1]†, Mohsan Jameel[1], Aamir Shafi[1] and Bryan Carpenter[2]

[1] SEECS, National University of Sciences and Technology (NUST), Pakistan
{bibrak.qamar, muhammad.ansar, mohsan.jameel, aamir.shafi}@seecs.nust.edu.pk
[2] School of Computing, University of Portsmouth, UK,
bryan.carpenter@port.ac.uk

Abstract

MPJ Express is a messaging system that allows computational scientists to write and execute parallel Java applications on High Performance Computing (HPC) hardware. The software is capable of executing in two modes namely cluster and multicore modes. In the cluster mode, parallel applications execute in a typical cluster environment where multiple processing elements communicate with one another using a fast interconnect like Gigabit Ethernet or other proprietary networks like Myrinet and Infiniband. In this context, the MPJ Express library provides communication devices for Ethernet and Myrinet. In the multicore mode, the parallel Java application executes on a single system comprising of shared memory or multicore processors. In this paper, we extend the MPJ Express software to provide two new communication devices namely the native and hybrid device. The goal of the native communication device is to interface the MPJ Express software with native—typically written in C—MPI libraries. In this setting the bulk of messaging logic is offloaded to the underlying MPI library. This is attractive because MPJ Express can exploit latest features, like support for new interconnects and efficient collective communication algorithms of the native MPI library. The second device, called the hybrid device, is developed to allow efficient execution of parallel Java applications on clusters of shared memory or multicore processors. In this setting the MPJ Express runtime system runs a single multithreaded process on each node of the cluster—the number of threads in each process is equivalent to processing elements within a node. Our performance evaluation reveals that the native device allows MPJ Express to achieve comparable performance to native MPI libraries—for latency and bandwidth of point-to-point and collective communications—which is a significant gain in performance compared to existing communication devices. The hybrid communication device—without any modifications at application level—also helps parallel applications achieve better speedups and scalability. We witnessed comparative performance for various benchmarks—including NAS Parallel Benchmarks—with hybrid device as compared to

*Native Device is contribution of Bibrak Qamar
†Hybrid Device is contribution of Ansar Javed

 Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2014

doi:10.1016/j.procs.2014.05.017

the existing Ethernet communication device on a cluster of shared memory/multicore processors.

Keywords: High Performance Computing; Java MPI; MPJ Express; Hybrid MPI; Native MPI for Java.

1 Introduction

The Message Passing Interface (MPI) standard [1] has become the *de facto* API for programming High Performance Computing (HPC) systems. These systems—as depicted in the recent TOP500 list—are built using multicore processors with multiple levels of caches and special accelerators like GPUs and FPGAs. These processors and accelerators are often connected through complex topology enabled by proprietary interconnects like Myrinet and Infiniband. In this context, most of the efforts of speeding up and scaling MPI programs have been geared towards conventional HPC programming languages, like C and Fortran, and messaging libraries such as MPICH [2] and Open MPI [3]. On the other hand, modern languages with features of object orientation, modularity, maintainability and portability have been treated with cynicism, mostly because of their poor computing performance and lack of high performance communication support [6]. This criticism is not justified anymore because most modern languages and their compilers and runtime environments have witnessed manifold performance improvements. An example of one such modern programming language is Java. By the use of Just-in-Time (JIT) compilers the performance gap between Java byte code and native code is becoming negligible [14]. The emergence of many popular and successful Java messaging libraries like mpiJava [8], FastMPJ [15] and MPJ Express [5] have successfully helped decrease communication gap between C/Fortran and Java applications on HPC hardware. The following Java-based projects and their success stories clearly depict that Java has become a competitive and scalable distributed systems programming language. One of the success stories is related to Hadoop, which is an open source Java implementation of the Google MapReduce framework used for scalable distributed processing of large data sets on clusters of computers. Another success is the recent resurrection of the mpiJava library by Open MPI users and developers community [16]. It indicates the viability of Java for HPC. Lastly, there is also interest in commercializing Java HPC libraries like Java Fast Sockets (JFS) and FastMPJ [4] under the umbrella of Torus Software Solutions.

MPJ Express is an MPI-like—implements the mpiJava 1.2 API—messaging library with an active user community. The software is capable of executing in two modes namely cluster and multicore modes. In the cluster mode, parallel applications execute in a typical cluster environment where multiple processing elements communicate with one another using a fast interconnect like Gigabit Ethernet or other proprietary networks like Myrinet and Infiniband. In this context, the MPJ Express library provides communication devices for Ethernet and Myrinet. In the multicore mode, the parallel Java application executes on a single system comprising of shared memory or multicore processors.

In this paper, we extend the MPJ Express software to provide two new communication devices namely the native and the hybrid device. The goal of the native communication device is to interface the MPJ Express software with native—typically written in C—MPI libraries. In this setting the bulk of messaging logic is offloaded to the underlying MPI library. This is attractive because MPJ Express can exploit latest features, like support for new interconnects and efficient collective communication algorithms of the native MPI library. With the addition of this new device, MPJ Express users have the option to either opt for portability—by using pure Java device—or performance—by using the native device. The second device, called

the hybrid device, is developed to allow efficient and transparent execution of parallel Java applications on clusters of shared memory or multicore processors. In this setting MPJ Express runtime system runs a single multithreaded process on each node of the cluster—the number of threads in each process is equivalent to processing elements within a node. Traditionally such clusters are programmed by writing hybrid parallel programs [9]—these applications use MPI for inter-node communication and use OpenMP (a shared memory API) for intra-node communication. In fact, we devised a similar technique in an earlier effort [12] where we used MPJ Express and Java OpenMP (JOMP) for inter-node and intra-node communication, respectively. However, this approach requires modifying the parallel application that can be avoided by using the hybrid communication device.

We evaluated the performance of the new devices, the native device and the hybrid device, by employing variety of performance tests including basic latency and bandwidth benchmarks for point-to-point and collective communication, Java NAS Parallel Benchmark (NPB), and Java Gadget-2 that is a real world scientific application for cosmological simulation. Our results indicate that the native device allows MPJ Express to achieve comparable performance to native MPI libraries—for latency and bandwidth of point-to-point and collective communications—which is a significant gain in performance compared to existing communication devices. The hybrid communication device—without any optimization at application level—also helps parallel applications achieve better speedups and scalability. We witnessed comparative performance for various benchmarks with hybrid device as compared to the existing Ethernet communication device on a cluster of shared memory/multicore processors.

This paper is organized as follows: Section 2 discusses related work; Section 3 describes the architecture of MPJ Express and where our new devices fit in; Section 4 and 5 describe the implementation of our native and hybrid device (also known as hybdev) respectively; Section 6 consists of performance evaluation of MPJ Express in comparison to other messaging systems and Section 7 concludes the paper.

2 Related Work

Early efforts at introducing Java bindings for MPI include the mpiJava 1.2 library [8] which provides MPI 1.1 functionality by using JNI wrappers on top of a native MPI library. This approach requires compiling the library for the target platform, which subsequently creates portability issues. Recently the Open MPI users and developers community has resurrected the mpiJava library [16]. Open MPI now provides Java bindings in its current release 1.7.4. In this setting the mpiJava can only use Open MPI as underlying message-passing library since it is integrated into the stack.

MPJ/Ibis [7] is non-thread safe implementation of MPJ API specifications on top of Ibis platform. The choice of Ibis makes MPJ/Ibis be deployed flexibly and efficiently on clusters made of propriety interconnects to Grid platforms.

FastMPJ [15] provides thread-safe pure Java devices for message-passing. It is implemented on mpiJava 1.2 specifications. FastMPJ is a propriety Java message-passing library that supports shared memory, TCP/IP and high throughput propriety interconnects like Infiniband.

MPJ Express [5] is an open source library that provides pure Java communication devices for shared memory multicore processors using Java threads, TCP/IP using Java NIO and high throughput interconnect like Myrinet. The MPJ Express architecture (discussed in detail in Section 3) allows it to use native MPI libraries for communication along with pure Java devices. MPJ Express uses JNI wrappers on top of native MPI library like mpiJava. In this way MPJ Express offloads bulk of communication logic to native MPI implementation. This setting is

primarily aimed at exploiting the optimizations offered by propriety MPI implementations for HPC systems connected with complex propriety topologies. Such functionality was not available in MPJ Express prior to this paper.

These modern HPC systems are mostly equipped with shared memory multicore processors. A combination of MPI and Open MP or Pthreads—*hybrid parallelism*—is used to efficiently communicate across network and shared memory, respectively. Java users achieve similar objectives by using MPJ Express or other message-passing libraries with JOMP [12]—OpenMP-like set of directives and library routine for Java.

Hybrid MPI [9] transparently exploit the hybrid parallelism from modern HPC systems. Hybrid MPI uses single threaded MPI processes with shared heap memory to communicate with in multicore processors on a single node. In the realm of Java, Ramos et al [11] presented first proof of concept for hybrid parallelism. They wrote a new communication device *smpniodev* by merging network and shared memory devices. Our approach towards extracting hybrid parallelism is different from them. Our hybrid device is built on top of network and shared memory devices. This design choice elevates the need for writing new hybrid devices for each network device.

To recap, our contribution—as compared to previous work—in this paper is twofold. 1) Enabling MPJ Express to use native MPI libraries, this helps harvest optimizations and innovations of native MPI implementations from within MPJ Express applications. 2) To provide hybrid communication device for MPJ Express to transparently exploit hybrid parallelism.

3 MPJ Express Architecture

MPJ Express employs a layered architecture that allows incremental development and provides flexibility to update layers in and out as needed. This flexibility alleviates constrains, as users can opt for using the pure Java device implementations or choose native MPI libraries. The architecture of MPJ Express is presented in Figure 1; it shows different levels of MPJ Express architecture stack: The MPJ API, high-level, base-level, *mpjdev* and *xdev*. The top three layers are exposed to the MPJ Express user to write distributed parallel applications. The base-level contains point-to-point communication primitives like `send` and `recv`. The high-level and the MPJ API contain collective communication routines, derived datatypes and virtual topologies.

The MPJ Express architecture has two device layers: the *mpjdev* layer and the *xdev* layer. The rationale behind introducing two device layers is to enable MPJ Express to call native MPI libraries. This is made available directly through the *mpjdev* layer. For pure Java devices *mpjdev* layer uses the second device layer—the *xdev*.

We made the *mpjdev* layer abstract and provided two concrete implementations. The first implementation, the *javampjdev* provides pure Java communication drivers focusing portability. The *javampjdev* in turn uses the second device layer the *xdev*. The second implementation of *mpjdev* layer, the *natmpjdev* uses JNI wrappers to a native MPI library. We call *natmpjdev* our *native* device and use the words interchangeably throughout this paper.

The *xdev* API [13] provides interface for developing new communication device drivers for underlying network interconnects. Currently, *xdev* layer contains device drivers for *smpdev* built on shared memory communication, *niodev* built on Java New IO (NIO), and *mxdev* built on Myrinet eXpress (MX) library. Our new *hybdev* device for MPJ Express implements the *xdev* API interface.

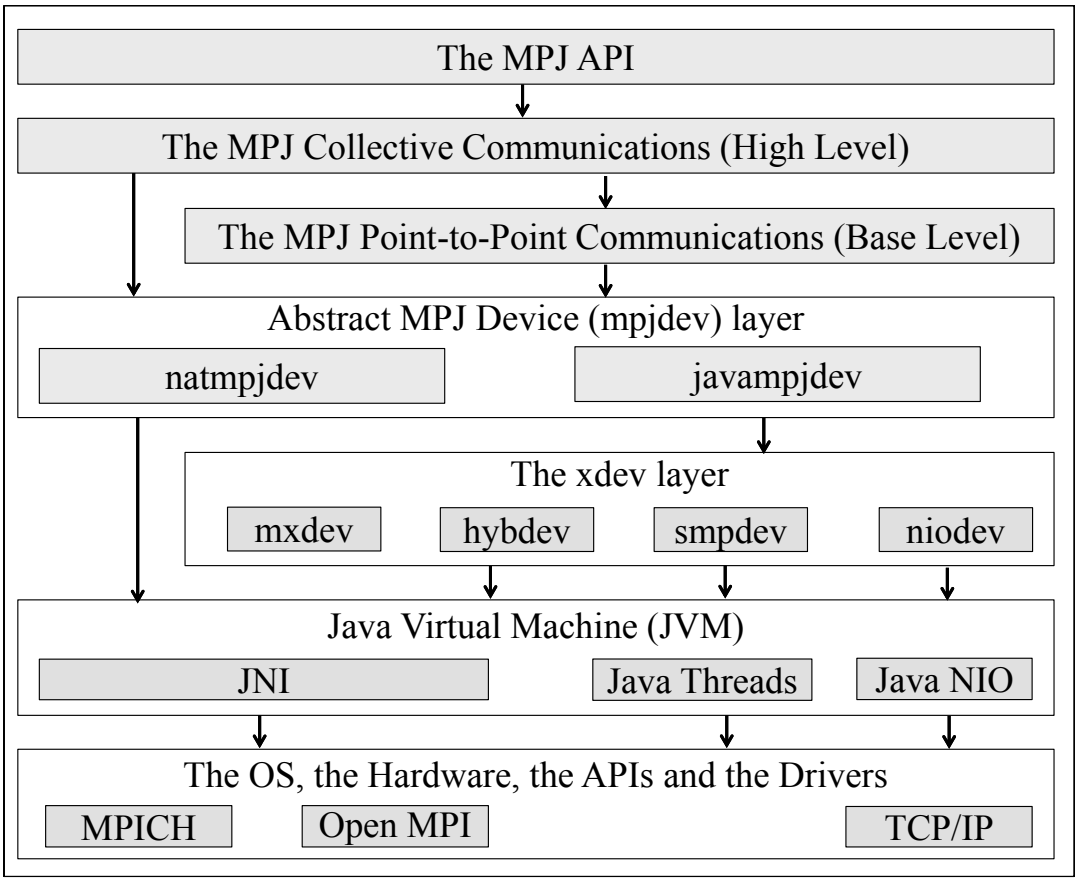

Figure 1: The MPJ Express Architecture.

4 Implementation of the Native Device

In this section we discuss implementation details of the *native* device. The *native* device implementation relies on *natmpjdev* that provides interface to native implementations of MPI using JNI wrappers. This work is inspired by the original mpiJava library, which uses JNI wrappers on top of native MPI library to provide object oriented Java MPI-like bindings. Such a native interface widens the usage of MPJ Express by providing flexibility to use multitude of network platforms and enables access to low level optimizations offered by propriety MPI implementations. This comes with two tradeoffs, portability and maintainability. The JNI wrapper library needs to be compiled for each target platform in order to deploy MPJ Express natmpjdev. The second tradeoff concerns the maintainability of the JNI wrapper library. Since the JNI wrapper library relies on native MPI implementations, compatibility needs to be insured between JNI wrapper library and MPI implementations in the future.

The development of *natmpjdev* required three major efforts: 1) We made the *mpjdev* layer abstract to provide a switch, at runtime, between *javampjdev* and our new *natmpjdev*. 2) To call native collective communication algorithms we enabled high-level layer to bypass base-level

layer. In this way *native* device interacts directly with the mpjdev layer to leverage native collective communications algorithms. Our *native* device also inherits the collective communications algorithms of MPJ Express. 3) We wrote a JNI wrapper library to communicate between MPJ Express Java code and native MPI C implementations.

MPJ Express programs using *native* device are launched through the native MPI program execution engine (commonly known as `mpirun`). This is needed for the MPJ Express to communicate with the underlying native MPI library. To understand this, consider a trivial example of *native* device initialization and finalization. The MPJ Express user application call to `MPI.Init()` translates into the native `MPI_Init()` using JNI. This initializes the universal communicator `COMM_WORLD`. We acquire the handle of `COMM_WORLD` that is subsequently used to acquire the handle of the underlying `Group`. The handles of `COMM_WORLD` and its `Group` are initialized in the *native* device. These handles are then used to initialize the values of `size` of the Communicator and `rank` of the process. Likewise the *native* device is finalized using the MPJ Express `MPI.Finalize()` method which through JNI wrapper library calls the `MPI_Finalize()` function.

For point-to-point communication our *native* device uses the MPJ Express intermediate buffering layer—called *mpjbuf*—to copy message data into an internal buffer. Currently MPJ Express has only one implementation of *mpjbuf* called `NIOBuffer`. Using `NIOBuffer` application data is copied as bytes into `ByteBuffer` and is sent to *mpjdev* layer. Java NIO provides functionality of allocating the `ByteBuffer` outside the Java Virtual Machine (JVM) memory using `allocateDirect()`. In this way the `ByteBuffer` memory is directly accessible to the *native* device. Our *native* device (JNI wrapper library) gets the address of this memory and passes it to the underlying native MPI library. It must be noted here that since we are communicating data using `ByteBuffer` the `MPI_BYTE` data type is used for all point-to-point communications.

In the case of non-blocking communication, we have extended the abstract `Request` class at the abstract *mpjdev* layer to create another abstract class called the `NativeRequest`. It holds the handle of `MPI_Request` returned by the native MPI library. Two classes: `NativeRecvRequest` and `NativeSendRequest` further extend `NativeRequest`, for non-blocking recv (`Irecv()`) and non-blocking send (`Isend()`) respectively. This was mainly needed to provide different set of functionality for the `Request.Wait()` method. For `Isend` operation the `Wait()` methods waits for the communication to finish and returns while for `Irecv` the `Wait()` method upon receiving the data copies the data in the *mpjbuf*.

5 Implementation of the Hybrid Device

This section discusses implementation of hybrid device based on *xdev* API. There are two possible design options for *hybdev* development. The first option is to develop an entirely new device by merging the source-code of *smpdev* with a network device like *niodev*. In fact, this approach was used to develop *smpniodev* by Ramos et al [11]. The second option is to rely on existing devices by creating their instances at runtime. The *hybdev* follows the second approach. The main reason is that this approach provides code reusability, which is crucial for software maintenance. The current implementation of *hybdev* supports *niodev* for inter-node communication.

5.1 Initialization and Finalization of Hybrid Device

The first challenge encountered during the implementation of our hybrid device was to maintain correct semantics for the parallel Java program while relying on two architecturally incompliant

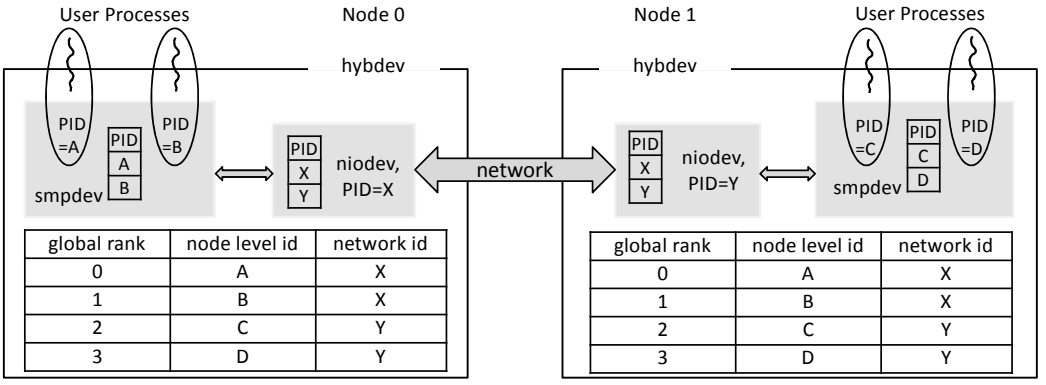

Figure 2: Initialization and communication mechanism in *hybdev*.

devices like *smpdev* and *niodev*. On one hand *smpdev* executes Java threads in a single JVM, while network device on the other hand acts as a conduit between JVMs running on distributed nodes. As discussed in our earlier work [13], *smpdev* is a multicore communication device where various threads represent different MPJ Express processes and each thread has `ProcessID` (PID) consisting of Universally Unique Identifier (UUID). One challenging requirement of *smpdev* is that it relies on shared variables between threads at the *xdev* layer; however user program variables must not be shared. *smpdev* and *hybdev* meet this requirement by relying on an intelligent class loading mechanism where different packages of the MPJ Express software are divided into two groups and then loaded with separate class loaders [13].

The user executes the parallel Java application by specifying the total number of processes and available nodes. Based on this, the MPJ Express runtime dynamically decides the number of threads to be executed on each node. Later, *hybdev* makes an instance of *niodev* that connects all JVMs across the network. In next step an instance of *smpdev* is created on each node. *smpdev* threads share the *niodev* instance for communication over the network. Each node in the network is identified by PID of *niodev* and this PID is used as network ID. *smpdev* threads across the network are identified by their PIDs and these PIDs are also called as thread IDs. As shown in Figure 2, MPI ranks (e.g. 0, 1 etc) in user space are translated to UUIDs in *mpjdev* layer of MPJ Express and then these UUIDs are mapped to the network IDs and thread IDs in *xdev* layer. For example in Figure 2, we depict execution of four parallel processes on two nodes of a cluster. In this case, each node has an instance of *smpdev* and *niodev*. The *smpdev* running on node 0 boots two threads—with PIDs A and B—representing user processes rank 0 and 1. Similarly, the *smpdev* on node 1 starts two threads—with PIDs C and D—representing user processes rank 2 and 3. In addition, an instance of *niodev* is booted with PIDs X and Y for nodes 0 and 1 respectively. In the last step *hybdev* integrates thread identifiers (A, B, C, and D) from *smpdev* instances across the network and develops a global set of identifiers—the table shown in Figure 2. In rank column of table numbers (0, 1 etc) are UUIDs of user threads as complete UUIDs are not mentioned for sake of simplicity. Using only one *niodev* per node allows MPJ Express to use more Java threads instead of processes and it substantially reduces the device initiation time.

5.2 Communication

Communication in *hybdev* is implemented using point-to-point message passing routines of underlying *niodev* and *smpdev*. Communication device selection in *hybdev* is based on network ID of source and destination of communicating processes. *hybdev* stores network ID of processes in a Java hashtable that has time complexity of O(1). For each `send` and `recv` request there will be one query to this hashtable. If source and destination are on same node the *hybdev* uses *smpdev* for communication otherwise *niodev* is used. To give an example, imagine that in Figure 2, rank 0 sends a message to rank 1. In this case *hybdev* gets the network ID of both source and destination of message and determines that it is intra-node communication; thus *smpdev* is used for message transfer. On the other hand, if rank 0 sends a message to the process with rank 3, *hybdev* checks the network IDs and determines that it is inter-node transfer so *niodev* is used. Messages reaching on a host are put into queues, and user invoke `MPI.Recv()` operation to de-queue a particular message. Messages in queues are distinguished on the basis of a key consisting of sourceID, tag and context. Both *smpdev* and *niodev* implement `RecvQueue` for keeping messages that are not completely received while user has posted `Recv()` method. `ArriveQueue` for keeping messages that have been received completely but `Recv()` method is not posted by user [13]. While *niodev* is shared between *smpdev* threads for network communication a problem of wrong receive occurs when two threads on the same node expect a message from single remote source. Any thread can de-queue message that was directed to the peer thread. This issue is tackled by adding destination process UUID into key that makes *niodev* capable of distinguishing between destination threads. Source and destination UUIDs of messages are provided in message header.

MPJ Express supports receiving wildcard messages with `MPI.ANY_SOURCE`. In *hybdev*, the first challenge is to decide which device to use for receiving such message. Currently we tackle this issue by launching a new on demand thread in *hybdev* that looks for the wildcard message in both *niodev* and *smpdev* queues and then message is de-queued when found in any of devices.

6 Performance Evaluation

In this section we evaluate the performance of our hybrid and native devices in MPJ Express and compare them with existing MPJ Express device (*niodev*) and other messaging systems. We begin by describing our test environment, which consisted of a 32 node cluster hosted at RCMS National University of Sciences and Technology, Pakistan. Each compute node contains two quad-core Intel Xeon E5520 processors with a main memory of 24G Bytes. The nodes are connected via Gigabit Ethernet and 40 Gbps QDR InfiniBand. Our software environment consisted of the Oracle JDK 1.7.0_25 version and GNU GCC 4.8.1. We integrated our new devices (*hybdev* and *native*) in the MPJ Express version 0.38, which we used as basis for our implementation. The native MPI library that we used was MPICH 3.0.4 for Gigabit Ethernet and MVAPICH2.2 for InfiniBand.

6.1 Point-to-Point and Collective Communication

Figure 3(a) and (b) show transfer time and throughput comparison across Gigabit Ethernet. The latency (transfer time for one byte) of 25.2 μsec of the C MPI library (MPICH3) is the lowest of all. MPJ Express with native device follows MPICH3 with 29.5 μsec. This is basically because the native device uses the same messaging mechanism of MPICH3. FastMPJ had a latency of 57 μsec. The MPJ Express *niodev* and *hybdev* that essentially uses that same pure

Java device (*niodev*) have slightly higher latency of 69.5 μsec and 79.1 μsec respectively

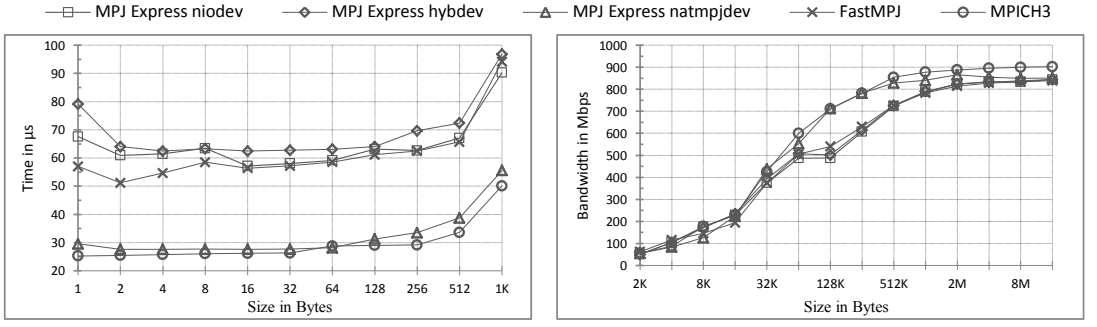

Figure 3: (a) Latency over Gigabit Ethernet; (b) Bandwidth over Gigabit Ethernet

Figure 3(b) shows that the throughput (bandwidth in Mbps) of MPICH3 is the highest. MPICH3 achieves 90% of maximum bandwidth for message size of 16M bytes. Again MPJ Express running with native device follows MPICH3 with a bandwidth of 85% of maximum bandwidth. The MPJ Express (*niodev* and *hybdev*) and FastMPJ attain a bandwidth of about 84% of maximum bandwidth.

Figure 4(a) and (b) show transfer time and throughput comparison across InfiniBand. MVAPICH2.2 has the lowest latency of 2.13 μsec whereas MPJ Express native has 4.90 μsec. Throughput (bandwidth in Gbps) of MVAPICH2.2 is the highest. MVAPICH2.2 achieves 22.63

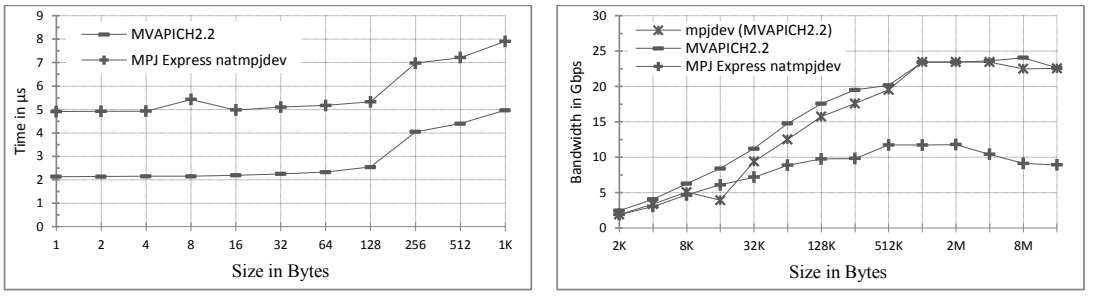

Figure 4: (a) Latency across InfiniBand; (b) Bandwidth across InfiniBand

Gbps for message size of 16M Bytes. MPJ Express running with native device suffers from performance loss with 8.91 Gbps for message size of 16M bytes. The reason for higher latency and low throughput of MPJ Express is a combination of the use of thread-safe algorithms and additional copying done on the user data. As described in Section 4, MPJ Express uses an intermediate buffering layer. This implies additional copying. When using the device layer—mpjdev—directly, the data has already been copied onto a direct `ByteBuffer`, the difference between the performances of MPJ Express and mpjdev in Figure 4(b) shows the overhead of packing (at sender) and unpacking (at receiver).

We evaluated the performance of collective communications primitives using the `Bcast()` operation and compared performance of three messaging systems for Gigabit Ethernet: MPJ Express running *niodev*, *hybdev* and native device, FastMPJ and MPICH3. For InfiniBand network we compared performance of our native device with MVAPICH2.2. We were also

interested in the performance of our native device using the MPJ Express collective communication algorithms. For this we configured MPJ Express native device to use Java collective communication routines and took results. We call this configuration "MPJ Express *natmpjdev J*". We performed our experiment varying message size from 16K bytes to 16M bytes on a total of 128 cores. The datatype we used for this experiment was `MPI.BYTE`. Figure 5(a) shows

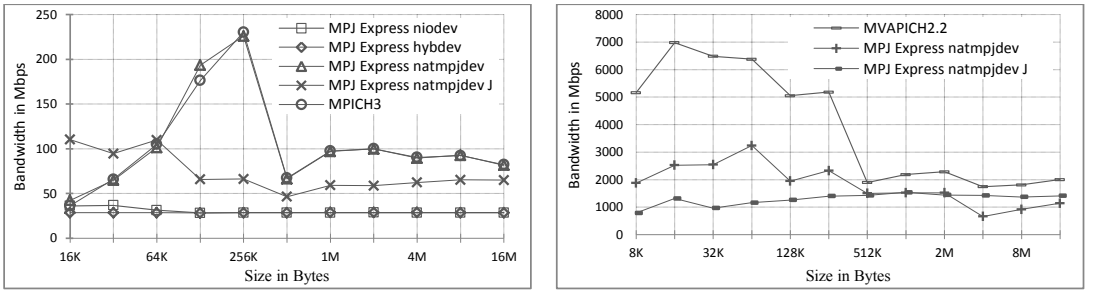

Figure 5: (a) `Bcast()` (128 Cores Gigabit Ethernet) (b) `Bcast()` (128 Cores InfiniBand)

results on Gigabit Ethernet where MPICH3 and MPJ Express running in native mode overall performed better with the highest bandwidth of 256 Mbps for message size 256K bytes and an average bandwidth of about 90 Mbps for message sizes between 512K bytes to 16M bytes. For the case of native device—using MPJ Express collectives, *niodev*, *hybdev* and FastMPJ an average bandwidth of 72 Mbps, 29.1 Mbps, 29 Mbps and 35.7 Mbps, respectively, was achieved for message sizes between 512K bytes to 16M bytes. Figure 5(b) shows the same trend for Infiniband where the native MPI library (MVAPICH2.2) leads with an average bandwidth of 1989.12 Mbps on 128 cores for message sizes between 512K bytes to 16M bytes whereas native device (*natmpjdev*) and native device with MPJ Express collectives (*natmpjdev J*) achieved 1212.69 Mbps and 1436.38 Mbps respectively.

6.2 Java NAS Parallel Benchmark

In this sub-section we evaluate the performance of Java NAS parallel benchmarks (NPB) kernels on Gigabit Ethernet. We chose workload Class B and C, which are relatively heavy workloads. The performance metric for the tests is Millions of Operations Per Second (MOPS), which refers to the measurement of kernel operations not the CPU operations used [10].

In the case of the CG kernel Figure 6(a) and Figure 6(b), MPJ Express *hybdev* and *niodev* overall perform better. For CG Class B hybdev and *niodev* achieve 2443.18 MOPS and 2460.69 MOPS respectively on 128 cores. MPJ Express native device and FastMPJ achieve 1743.53 MOPS and 1529.57 MOPS respectively on 128 cores. CG Class C also shows the same trend with *hybdev* and *niodev* outperforming. *hybdev* gains significant improvement than *niodev* (average 22.2%) for 32 and 64 cores. The performance drop of *hybdev* on 128 cores is mainly because the cores on a single node share the same network device creating congestion. One way to improve the performance of *hybdev* is to launch multiple network devices on a single node. This will allow the cores not to compete for a single network device instance, which creates congestion. MPJ Express native device and FastMPJ achieve 2415.22 MOPS and 1824.10 MOPS, respectively, on 128 cores for CG Class C. The EP kernel owing to its embarrassingly parallel nature scales very well for all devices, as shown in Figure 6(c) and Figure 6(d). MPJ Express *niodev* performs better with 1013.13 MOPS on 128 cores for Class B. FastMPJ and MPJ Express *hybdev*, *natmpjdev* achieve 880.73 MOPS, 878.23 MOPS and 727.87 MOPS, respectively.

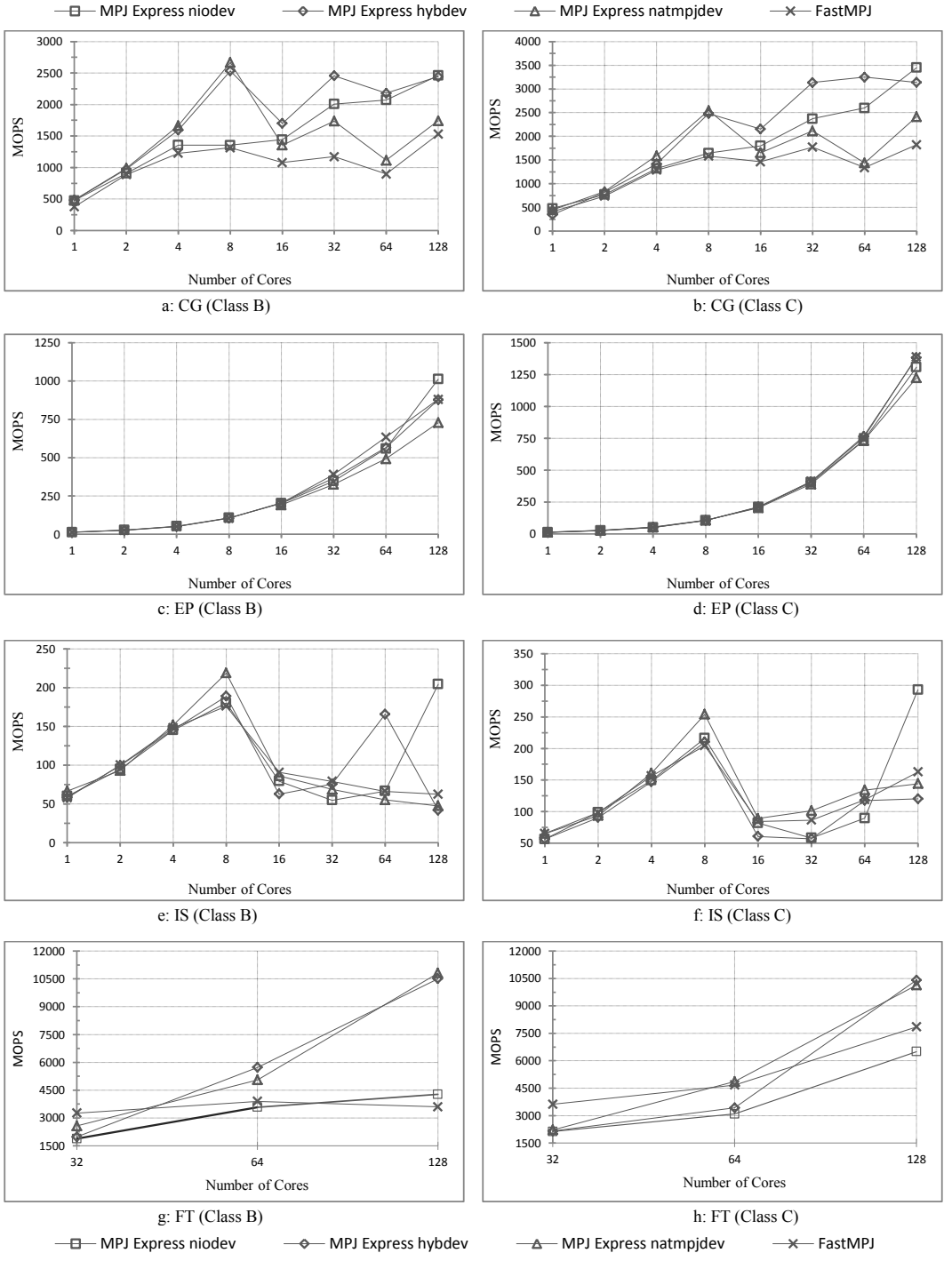

Figure 6: Java NAS Parallel Benchmarks

For Cass C FastMPJ and MPJ Express *hybdev* show comparative performance of an average 1380 MOPS followed by *niodev* and *natmpjdev* with 1309.06 MOPS and 1224.73 MOPS respectively on 128 cores.

The IS kernel (Figure 6 e and f), which does Integer Sort performed better on single node i.e. up until 8 cores but later improved performance as the cores increased. For Class B, on a total of 64 cores MPJ Express *hybdev* performed better with 165.77 MOPS, which is an improvement of 59.7% as compared to *niodev*. On higher number of cores (i.e. 128) *hybdev* with 41.83 MOPS suffer performance loss of about 80% as compared to *niodev* (204.47 MOPS). FastMPJ and *natmpjdev* achieved 62.61 MOPS and 47.55 MOPS respectively. Figure 8(b) Class C shows the same trend with *niodev* achieving highest MOPS of 292.90 followed by FastMPJ with 162.74 MOPS. *natmpjdev* and *hybdev* achieve 144.24 and 120.20 MOPS respectively.

For FT kernel the MPJ Express *natmpjdev* and *hybdev* achieve the highest performance of 10810.99 MOPS and 10507.44 MOPS respectively on a total of 128 cores for Class B, Figure 6(g). Again for Class C Figure 6(h) *hybdev* and *natmpjdev* outperform others with 10408.72 MOPS and 10137.72 MOPS on 128 cores. For Class B *niodev* achieved 4270.89 MOPS followed by FastMPJ with 3599.08 MOPS. For Class C FastMPJ achieved 7850.09 MOPS followed by 6494.99 MOPS. *hybdev* gains performance improvement of 59.3% and 37.6% for Class B and Class C respectively as compared to *niodev*. A reason for MPJ Express *hybdev* and *natmpjdev* significantly outperforming FastMPJ and *niodev* is that the formers better exploit data locality. The FT kernel splits the MPI communicator and performs communication on subsets of the total cores. This way *hybdev* and *natmpjdev* achieve better performance by using shared memory communication most of the time. It must be mentioned here that we were unable to run for 2, 4, 8 and 16 cores because of memory constraints.

6.3 Java Gadget-2

Gadget-2 is a cosmological N-body and hydrodynamics simulation code. Here we present results of Java version of the Gadget-2. Our previous paper contains details on this [12]. For our performance evaluation we used a cluster formation problem containing 276498 particles. As shown in Figure 7 our *hybdev* completed the simulation in 19.2 minutes followed by FastMPJ with 19.6 minutes on 64 cores. *hybdev* achieved a performance improvement of 21.5% as compared to *niodev*, which took 23.3 minutes to complete the simulation on 64 cores. The native device initially performed better but on 64 cores took 23.23 minutes.

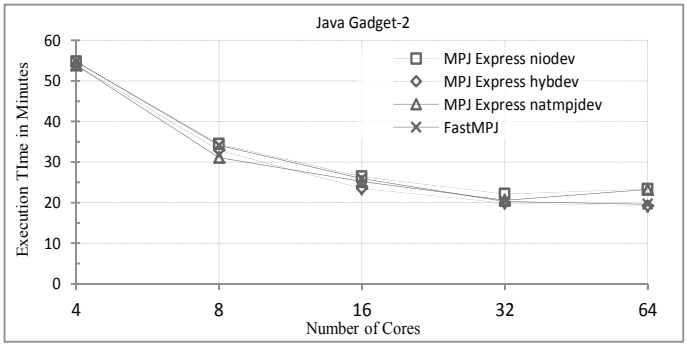

Figure 7: Execution time in minutes of Java Gadget-2

7 Conclusions and Future Work

This paper presented two new communication devices for MPJ Express to improve scalability of parallel Java applications on modern HPC systems. In particular we developed—*native device*—for using native MPI libraries from within MPJ Express programs and—*hybdev*—for clusters with shared memory and multicore processors. With the addition of this new device, MPJ Express users have the option to either opt for portability—by using pure Java device—or performance—by using the native device. The second device, *hybdev*, is developed to allow efficient and transparent execution of parallel Java applications on clusters of shared memory or multicore processors.

To help us understand the performance of our new devices we evaluated them with existing devices and other messaging libraries. We evaluated our devices using basic latency and bandwidth benchmarks for point-to-point and collective communication, Java NAS Parallel Benchmark (NPB), and Java Gadget-2 that is a real world scientific application for cosmological simulations. Our performance evaluation reveals that the native device allows MPJ Express to achieve comparable performance to native MPI libraries—for latency and bandwidth of point-to-point and collective communications—which is a significant gain in performance compared to existing communication devices. The hybrid communication device—without any modifications at application level—also helps parallel applications achieve better speedups and scalability. We witnessed comparative performance for various benchmarks—NAS Parallel Benchmarks—with hybrid device as compared to the existing Ethernet communication device on a cluster of shared memory/multicore processors.

The newly developed MPJ Express communication devices have created several new opportunities for the Java HPC community. With the *native device* it is now easier to transition to MPI-2 and MPI-3 features like one-sided communication, dynamic process management, non-blocking and neighborhood collective operations. In the context of *hybdev*, we plan to investigate efficient collective communication algorithms that exploit locality of processes on clusters built with shared memory and multicore processors.

References

[1] Message Passing Interface specifications. `http://www.mpi-forum.org/docs/docs.html/`, 2014. [Online; accessed 27-March-2014].

[2] MPICH. `http://www.mpich.org/`, 2014. [Online; accessed 27-March-2014].

[3] Open MPI. `http://www.open-mpi.org/`, 2014. [Online; accessed 27-March-2014].

[4] Torus Software Solutions. `http://torusware.com/`, 2014. [Online; accessed 27-March-2014].

[5] Mark Baker, Bryan Carpenter, and Aamir Shafi. MPJ Express: Towards Thread Safe Java HPC. In *IEEE International Conference on Cluster Computing (Cluster 2006)*, pages 1–10. IEEE, Sept 2006.

[6] Brian Blount and Siddhartha Chatterjee. An Evaluation of Java for Numerical Computing. In *Proceedings of Second International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, pages 35–46. Springer, 1998.

[7] Markus Bornemann, Rob V. Nieuwpoort, and Thilo Kielmann. MPJ/Ibis: A Flexible and Efficient Message Passing Platform for Java. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3666 of *Lecture Notes in Computer Science*, pages 217–224. Springer, 2005.

[8] Bryan Carpenter, Geoffery Fox, Sung-Hoon Ko, and Sang Lim. mpiJava 1.2: API Specification. Technical report, Northeast Parallel Architectures Center, Syracuse University, October 1999. http://www.hpjava.org/reports/mpiJava-spec/mpiJava-spec/mpiJava-spec.html.

[9] Andrew Friedley, Greg Bronevetsky, Torsten Hoefler, and Andrew Lumsdaine. Hybrid MPI: Efficient Message Passing for Multi-core Systems. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 18:1–18:11. ACM, 2013.

[10] Damin A. Mallón, Guillermo L. Taboada, Juan Touriño, and Ramón Doallo. NPB-MPJ: NAS Parallel Benchmarks Implementation for Message-Passing in Java. In *17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 181–190. IEEE, Feb 2009.

[11] Sabela Ramos, Guillermo L. Taboada, Juan Touriño, and Ramón Doallo. Scalable Java Communication Middleware for Hybrid Shared/Distributed Memory Architectures. In *IEEE 13th International Conference on High Performance Computing and Communications (HPCC)*, pages 221–228. IEEE, Sept 2011.

[12] Aamir Shafi, Bryan Carpenter, and Mark Baker. Nested parallelism for multi-core HPC systems using Java. *J. Parallel Distrib. Comput.*, 69(6):532–545, 2009.

[13] Aamir Shafi, Jawad Manzoor, Kamran Hameed, Bryan Carpenter, and Mark Baker. Multicore-enabling the MPJ Express Messaging Library. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 49–58. ACM, 2010.

[14] Guillermo L. Taboada, Sabela Ramos, Roberto R. Expósito, Juan Touriño, and Ramón Doallo. Java in the High Performance Computing Arena: Research, Practice and Experience. *Sci. Comput. Program.*, 78(5):425–444, May 2013.

[15] Guillermo L. Taboada, Juan Touriño, and Ramón Doallo. F-MPJ: Scalable Java Message-passing Communications on Parallel Systems. *J. Supercomput.*, 60(1):117–140, April 2012.

[16] Oscar Vega-Gisbert, Jose E. Roman, Siegmar Groß, and Jeffrey M. Squyres. Towards the Availability of Java Bindings in Open MPI. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 141–142. ACM, 2013.