J Supercomput (2012) 61:141–165
DOI 10.1007/s11227-011-0654-9

Design of scalable Java message-passing communications over InfiniBand

Roberto R. Expósito · Guillermo L. Taboada · Juan Touriño · Ramón Doallo

Published online: 23 July 2011

Abstract This paper presents `ibvdev` a scalable and efficient low-level Java message-passing communication device over InfiniBand. The continuous increase in the number of cores per processor underscores the need for efficient communication support for parallel solutions. Moreover, current system deployments are aggregating a significant number of cores through advanced network technologies, such as InfiniBand, increasing the complexity of communication protocols, especially when dealing with hybrid shared/distributed memory architectures such as clusters. Here, Java represents an attractive choice for the development of communication middleware for these systems, as it provides built-in networking and multithreading support. As the gap between Java and compiled languages performance has been narrowing for the last years, Java is an emerging option for High Performance Computing (HPC).

The developed communication middleware `ibvdev` increases Java applications performance on clusters of multicore processors interconnected via InfiniBand through: (1) providing Java with direct access to InfiniBand using InfiniBand Verbs API, somewhat restricted so far to MPI libraries; (2) implementing an efficient and scalable communication protocol which obtains start-up latencies and bandwidths similar to MPI performance results; and (3) allowing its integration in any Java parallel and distributed application. In fact, it has been successfully integrated in the Java messaging library MPJ Express.

R.R. Expósito · G.L. Taboada (✉) · J. Touriño · R. Doallo
Computer Architecture Group, Dept. of Electronics and Systems, University of A Coruña, A Coruña, Spain
e-mail: taboada@udc.es

R.R. Expósito
e-mail: rreye@udc.es

J. Touriño
e-mail: juan@udc.es

R. Doallo
e-mail: doallo@udc.es

The experimental evaluation of this middleware on an InfiniBand cluster of multi-core processors has shown significant point-to-point performance benefits, up to 85% start-up latency reduction and twice the bandwidth compared to previous Java middleware on InfiniBand. Additionally, the impact of `ibvdev` on message-passing collective operations is significant, achieving up to one order of magnitude performance increases compared to previous Java solutions, especially when combined with multithreading. Finally, the efficiency of this middleware, which is even competitive with MPI in terms of performance, increments the scalability of communications intensive Java HPC applications.

Keywords Message-Passing in Java (MPJ) · InfiniBand · Multicore architectures · High performance computing · Remote Direct Memory Access (RDMA) · Performance evaluation

1 Introduction

Java is the leading programming language both in academia and industry environments, and it is an emerging alternative for High Performance Computing (HPC) [1] due to its appealing characteristics: built-in networking and multithreading support, object orientation, automatic memory management, platform independence, portability, security, an extensive API, and a wide community of developers. Furthermore, in the era of multicore processors, the use of Java threads is considered a feasible option to harness the performance of these processors.

Java initially was severely criticized for its poor computational performance [2], but the performance gap between Java and native (compiled) languages like C or Fortran has been narrowing for the last years. The main reason is that the Java Virtual Machine (JVM), which executes Java applications, is now equipped with Just-in-Time (JIT) compilers that obtain native performance from Java bytecode. Nevertheless, the tremendous improvement in its computational performance is not enough for Java to be a successful language in the area of parallel computing, as the performance of the communications is also essential to achieve high scalability in Java for HPC.

Message-passing is the most widely used parallel programming paradigm as it is highly portable, scalable, and usually provides good performance. It is the preferred choice for parallel programming distributed memory systems such as multi-core clusters, currently the most popular system deployments due to their scalability, flexibility, and interesting cost/performance ratio. Here, Java represents an attractive alternative to languages traditionally used in HPC, such as C or Fortran, for the development of applications for these systems as it provides built-in networking and multi-threading support, key features for taking full advantage of hybrid shared/distributed memory architectures. Thus, Java can use threads in shared memory (intranode) and its networking support for distributed memory (internode) communications.

The increasing number of cores per system demands efficient and scalable message-passing communication middleware. However, up to now Message-Passing in Java (MPJ) implementations have been focused on providing portable communication devices, rather than concentrate on developing efficient low-level communication devices on high-speed networks. The lack of efficient support for high-speed networks in Java, due to its inability to control the underlying specialized hardware, results in lower performance than MPI, especially for short messages. This paper presents a scalable and efficient Java low-level message-passing communication device, `ibvdev`, aiming to its integration in MPJ implementations in order to provide higher performance on InfiniBand multicore clusters. In fact, it has been already integrated successfully in the MPJ library MPJ Express [3] (http://mpj-express.org).

The structure of this paper is as follows: Sect. 2 presents InfiniBand background information. Section 3 introduces the related work. Section 4 describes the design and implementation of the efficient `ibvdev` middleware, covering in detail the operation of the communication algorithms that provide the highest performance over InfiniBand. Section 5 shows the performance results of the implemented library on an InfiniBand multicore cluster. The evaluation consists of a micro-benchmarking of point-to-point and collectives primitives, as well as a kernel/application benchmarking in order to analyze the impact of the use of the library on their overall performance. Section 6 summarizes our concluding remarks.

2 Java communications over InfiniBand

2.1 InfiniBand architecture

The InfiniBand Architecture (IBA) [4] defines a System Area Network (SAN) for interconnecting processing nodes and I/O nodes. In an InfiniBand network, processing nodes and I/O nodes are connected to the fabric by Channel Adapters (CA). Channel Adapters usually have programmable DMA engines with protection features. There are two kinds of channel adapters: Host Channel Adapter (HCA) and Target Channel Adapter (TCA). HCAs sit on processing nodes and TCAs connect I/O nodes to the fabric.

The InfiniBand communication stack consists of different layers. The interface presented by channel adapters to consumers belongs to the transport layer. A queue-based model is used in this interface. A Queue Pair (QP) in InfiniBand Architecture consists of two queues: a send queue and a receive queue. The send queue holds instructions to transmit data and the receive queue holds instructions that describe where received data has to be placed. Communication operations are described in Work Queue Requests (WQR), or descriptors, and submitted to the work queue. Once submitted, a Work Queue Request becomes a Work Queue Element (WQE). WQEs are executed by Channel Adapters. The completion of work queue elements is reported through Completion Queues (CQs). Once a work queue element is finished, a completion entry is placed in the associated completion queue. Applications can check the completion queue to see if any work queue request has been finished.

2.1.1 Channel and memory semantics

InfiniBand Architecture supports both channel and memory semantics. In channel semantics, send/receive operations are used for communication. To receive a message, the programmer posts a receive descriptor which describes where the message should be put at the receiver side. At the sender side, the programmer initiates the send operation by posting a send descriptor. The send descriptor describes where the source data is but does not specify the destination address at the receiver side. When the message arrives at the receiver side, the hardware uses the information in the receive descriptor to put data in the destination buffer. Multiple send and receive descriptors can be posted and they are consumed in FIFO order. The completion of descriptors are reported through CQs.

In memory semantics, Remote Direct Memory Access (RDMA) write and RDMA read operations are used instead of send and receive operations. These operations are one-sided and do not incur software overhead at the other side. The sender initiates RDMA operations by posting RDMA descriptors. A RDMA descriptor contains both the local data source address and the remote data destination address. At the sender side, the completion of a RDMA operation can be reported through CQs. The operation is transparent to the software layer at the receiver side.

Both communication semantics require communication memory to be registered with InfiniBand hardware and pinned in memory. The registration operation involves informing the network-interface of the virtual to physical address translation of the communication memory. The pinning operation requires the operating system to mark the pages corresponding to the communication memory as non-swappable. Thus, communication memory stays locked in physical memory, and the network-interface can access it as desired.

2.1.2 Transport services

There are five transport modes defined by the InfiniBand specification: Reliable Connection (RC), eXtended Reliable Connection (XRC), Reliable Datagram (RD), Unreliable Connection (UC), and Unreliable Datagram (UD). All transports provide a checksum verification.

Reliable Connection (RC) is the most popular transport service for implementing MPI over InfiniBand. As a connection-oriented service, a QP with RC transport must be dedicated to communicating with only one other QP. A process that communicates with N other peers must have at least N QPs created. The RC transport provides almost all the features available in InfiniBand, most notably reliable send/receive, RDMA and atomic operations.

RC transport makes no distinction between connecting a process (generally one per core for MPI) and connecting a node. Thus, the associated resource consumption increased directly in relation to the number of cores in the system. To address this problem eXtended Reliable Connection (XRC) was introduced. Instead of having a per-process cost, XRC was designed to allow a single connection from one process to an entire node. XRC provides the services of the RC transport, but defines a very different connection model and method for determining data placement on the receiver in channel semantics. When using the RC transport, the connection model is purely based on processes. By contrast, XRC allows connection optimization based on the location of a process. The node of the peer to connect to is now taken into account, so instead of requiring a new QP for each process, now each process only needs to have

Table 1 Operations available for each transport service

Operation	RC	XRC	UC	RD	UD
Send (with immediate)	X	X	X	X	X
Receive	X	X	X	X	X
RDMA write (with immediate)	X	X	X	X	
RDMA read	X	X		X	
Atomic	X	X		X	

one QP per node to be fully connected. This reduces the number of QPs required by a factor of the number of cores per node.

Unreliable Connection (UC) provides a connection-oriented service with no guarantees of ordering or reliability. It supports RDMA write capabilities and sending messages larger than the Maximum Transmission Unit (MTU) size. Being connection-oriented in nature, every communicating peer requires a separate QP. In regard to resources required, it is identical to RC, while no providing reliable service. Thus, it appears unattractive for implementing MPI over this transport.

Unreliable Datagram (UD) is a connection-less and unreliable transport, the most basic transport specified for InfiniBand. As a connection-less transport, a single UD QP can communicate with any number of other UD QPs. However, the UD transport has a number of limitations. The UD transport does not provide any reliability: lost packets are not reported and the arrival order is not guaranteed. However, this can be solved relying on Reliable Datagram (RD). Moreover, UD transport does not enable RDMA. All communication must be performed using channel semantics, i.e., send/receive.

Table 1 shows the available operations for each transport service, since not all transport services support all operations, which has to be taken into account for a message-passing middleware implementation.

2.1.3 Shared receive queues

Shared Receive Queues (SRQs) were introduced in the InfiniBand 1.2 specification to address scalability issues with InfiniBand memory usage. In order to receive a message on a QP, a receive buffer must be posted in the Receive Queue (RQ) of that QP. To achieve high-performance, MPI implementations prepost buffers to the RQ to accommodate unexpected messages. When using the RC transport of InfiniBand, one QP is required per communicating peer. However, this task of preposting receives on each QP can have very high memory requirements for communication buffers. Recognizing that such buffers could be pooled, SRQ support was added so instead of connecting a QP to a dedicated RQ, buffers could be shared across QPs. In this method, a smaller pool can be allocated and then refilled on demand instead of preposting on each connection.

2.2 Message-passing communication devices

Message-passing libraries usually support new transport protocols through the use of pluggable low-level communication devices, such as Abstract Device Interface (ADI)

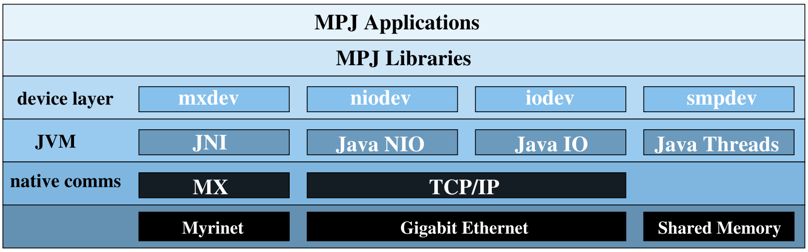


Fig. 1 Communications support of MPJ applications

in MPICH, Byte Transfer Layer (BTL) in OpenMPI, and `xdev` [5] in MPJ Express. These communication devices abstract the particular operation of a communication protocol, such Myrinet eXpress (MX), uDAPL (user Direct Access Programming Library), InfiniBand Verbs (IBV), Shared Memory, or SCTP (Stream Control Transmission Protocol), conforming to an API on top of which the message-passing library implements its communications.

Figure 1 presents an overview of the communications support of MPJ applications on the high-speed Myrinet network, on Gigabit Ethernet, and on shared memory. From top to bottom, MPJ applications rely on MPJ libraries, whose communication support is implemented in the device layer. Current Java communication devices are implemented either on JVM threads (`smpdev`, a multithreading device), on sockets over the TCP/IP stack (`niodev` on Java NIO sockets and `iodev` on Java IO sockets), or on native communication layers such as Myrinet eXpress (`mxdev`, a device on MX).

Regarding InfiniBand, up to now no direct support was made available for MPJ applications to fully exploit the communication capability of InfiniBand networks. This lack of direct InfiniBand support in Java requires the use of upper layer protocols such as IPoIB [6] (IP over InfiniBand) TCP emulation, as shown in Fig. 2, or SDP (Sockets Direct Protocol), the high performance native sockets library on InfiniBand. However, the use of IPoIB, the only communication library that fully supports Java over InfiniBand, shows quite poor performance [7]. Moreover, when relying on SDP the performance generally improves, but this is not always possible. Regarding MPI libraries, their direct InfiniBand support has been implemented some years ago on top of InfiniBand Verbs (IBV) API (see Fig. 2), achieving very high performance results. Therefore, our objective is the implementation of the direct InfiniBand support in Java on IBV through the development of a low-level Java communication device that can take advantage of InfiniBand RDMA transfers, thus outperforming significantly previous Java support on InfiniBand.

3 Related work

Current research on efficient Java communication libraries over InfiniBand is, to our knowledge, restricted to Jackal, Aldeia, Java Fast Sockets (JFS), Jdib, and uStream projects, next presented. Jackal [8] is a Java DSM (Distributed Shared Memory) middleware for clusters with InfiniBand Verbs support, embracing also RDMA transfers,

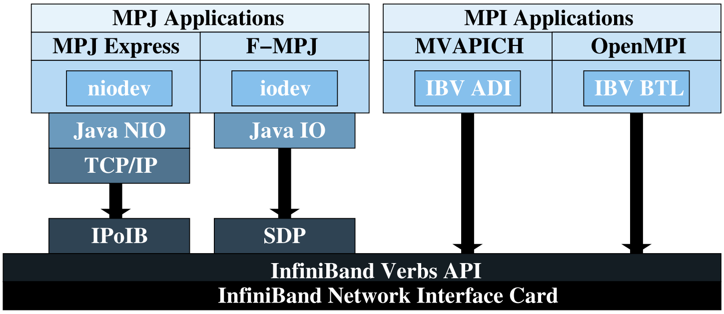


Fig. 2 MPI/MPJ applications support on InfiniBand

but it does not provide any API to Java developers as it only implements data transfers specifically for Jackal. Aldeia [9] is a proposal of an asynchronous sockets communication layer over InfiniBand whose preliminary results were encouraging, but requires an extra-copy, which incurs an important overhead to provide asynchronous write operations, whereas the read method is synchronous.

JFS [10] is our high performance Java socket implementation for efficient shared memory and high-speed networks support. JFS relies on SDP (see Fig. 2) to support Java communication over InfiniBand. Moreover, JFS avoids the need for primitive data type array serialization and reduces buffering and unnecessary copies. Nevertheless, the use of the sockets API is a significant drawback to support efficient message-passing communications.

Jdib [11, 12] (Java Direct InfiniBand) is a Java encapsulation of IBV API which maximizes Java communication performance using directly, through Java Native Interface (JNI), the InfiniBand RDMA mechanism. The main contribution of Jdib is its direct access to RDMA, providing to performance-concerned developers, for the first time, a Java RDMA API. Thus, Jdib significantly outperforms its alternatives, currently limited to IPoIB- and SDP-based solutions. The main drawbacks of Jdib are its low-level API and the JNI overhead incurred for each Jdib operation.

uStream [13] is a user-level stream protocol implemented on top of IBV that provides a higher level API than Jdib. In fact, uStream abstracts developers from the most tedious operations in Jdib, such as the buffer management, synchronization and the use of the IBV API, while fully exploiting InfiniBand RDMA performance. Therefore, uStream is much more effective and easier to use than Jdib for building parallel and distributed applications.

4 ibvdev: efficient Java communications over InfiniBand

This section presents the design and implementation of the `ibvdev` communication device, the Java message-passing middleware over InfiniBand developed in this paper. Unlike VIA [14, 15], InfiniBand architecture does not specify an API. Instead, it defines the functionality provided by HCAs to operating systems in terms of Verbs (a "verb" is a semantic description of a function that must be provided). The Verbs interface specifies such functionality as transport resource management, multicast, work request processing, and event handling. The most important implementation used today of Verbs interface is the IBV API provided by the OFED (OpenFabrics Enterprise Distribution) driver distributed by the OpenFabrics Alliance [16]. IBV is also the lowest level InfiniBand networking API for applications, available only in C language. Therefore, any Java communication support on IBV must resort to JNI in order to access IBV API and obtain the best possible performance, the target of the communication middleware developed, `ibvdev`.

4.1 Message-passing in Java libraries

There have been several efforts [1] over the last decade to develop a Java message-passing system since its introduction [17]. Most of these projects were prototype implementations, without maintaining. Currently, the most relevant ones in terms of uptake by the HPC community are mpiJava [18], MPJ Express [3], MPJ/Ibis [19] and F-MPJ [20].

mpiJava [18] is a Java messaging system that uses JNI to interact with the underlying native MPI library. This project has been perhaps the most successful Java HPC messaging system, in terms of uptake by the community. However, although its performance is usually high, mpiJava currently only supports some native MPI implementations, as wrapping a wide number of functions and heterogeneous runtime environments entails an important maintaining effort. Additionally, this implementation presents instability problems, derived from the native code wrapping (all MPJ methods are wrapped), and has thread safety issues in the wrapper layer, being unable to take advantage of multicore systems through multithreading, even if the underlying MPI library is thread safe.

MPJ Express is an MPJ implementation of the mpiJava 1.2 API [17] specification. MPJ Express is thread-safe and presents a modular design which includes a pluggable architecture of communication devices that allows to combine the portability of the "pure" Java New I/O package (Java NIO) communications (`niodev` device) with the high performance Myrinet support (through the native Myrinet eXpress communication library in the `mxdev` device).

MPJ/Ibis [19] is an implementation of the JGF MPJ API [21] specification on top of Ibis [22]. 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 nonblocking communication at the device level, the higher-levels only use blocking versions of these methods. Nevertheless, MPJ/Ibis does not provide a multithreaded communication device, unlike MPJ Express, key to harness the performance of multicore processors.

F-MPJ [20] is our message-passing communication middleware that provides shared memory and high-speed networks (e.g., InfiniBand, Myrinet, and SCI) communication support through the use of JFS. However, the use of Java IO sockets in its communication device `iodev` limits scalability as the progress engine of F-MPJ has to check every connection for incoming messages, unlike Java NIO sockets whose support is already implemented in the `select` method.

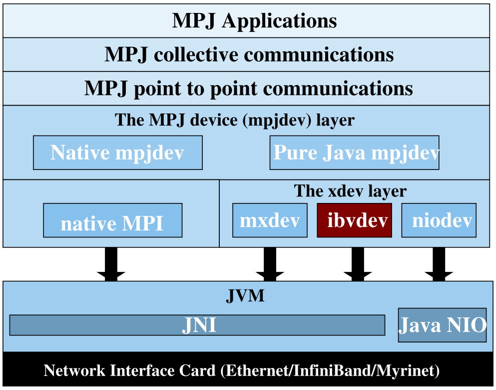


Fig. 3 Overview of the MPJ Express design including `ibvdev`

MPJ Express project is currently the most active project in terms of adoption by the HPC community, presence on academia and production environments, and available documentation. This project is also stable and publicly available along with its source code at http://mpj-express.org. Therefore, MPJ Express has been selected for the integration of the `ibvdev` middleware in a production MPJ library.

4.2 MPJ Express communication devices design

MPJ Express has a layered design that enables its incremental development and provides the capability to update and swap layers in or out as required. Thus, at runtime end users can opt to use a high performance proprietary network device, or choose a pure Java device, based either on sockets or threads, for portability.

Figure 3 illustrates an overview of the MPJ Express design and the different levels of the software. From top to bottom, it can be seen that a message-passing application in Java (MPJ application) calls MPJ Express point-to-point and collective primitives. These primitives implement the MPJ communications API on top of the `xdev` layer, which has been designed as a pluggable architecture and provides a simple but powerful API. This design facilitates the development of new communication devices in order to provide custom implementations on top of specific native libraries and HPC hardware. Thus, `xdev` is portable as it presents a single API and provides efficient communication on different system configurations.

Figure 3 also shows the three implementations of the `xdev` API for networked communication: `niodev` on Java NIO, and hence TCP/IP, and `mxdev` on Myrinet, as well as the developed `xdev` middleware for direct InfiniBand support, `ibvdev` (depicted in red).

4.2.1 `xdev` *API design*

The `xdev` API, presented in Listing 1, has been designed with the goal of being simple and small, providing only basic communication methods, in order to ease the development of `xdev` devices. An `xdev` communication device is similar to the MPI communicator class, but with reduced functionality. The `init` method starts the communication device operation. The `id` method returns the identification (`ProcessID`) of the device. The `finish` method is the last method to be called and completes the device operation.

The `xdev` communication primitives only include point-to-point communication, both blocking (`send` and `recv`, like MPI_Send and MPI_Recv) and nonblocking (`isend` and `irecv`, like MPI_Isend and MPI_Irecv). Synchronous communications are also embraced (`ssend` and `issend`). These communication methods use `PID` (ProcessID) objects instead of using ranks as arguments to send and receive primitives. In fact, the `xdev` layer is focused on providing basic communication methods and it does not deal with high level message-passing abstractions such as groups and communicators. Therefore, a `PID` object unequivocally identifies a device object.

```
public abstract class Device {
public static Device newInstance(String dev);
ProcessID[] init(String[] args);
ProcessID id();
void finish();

Request isend(Buffer buf, PID dest, int tag, int cntx);
void send(Buffer buf, PID dest, int tag, int cntx);
Request issend(Buffer buf, PID dest, int tag, int cntx);
void ssend(Buffer buf, PID dest, int tag, int cntx);
Status recv(Buffer buf, PID src, int tag, int cntx);
Request irecv(Buffer buf, PID src, int tag, int cntx, Status s);
Status probe(PID src, int tag, int cntx);
Status iprobe(PID src, int tag, int cntx);
Request peek();
}
```

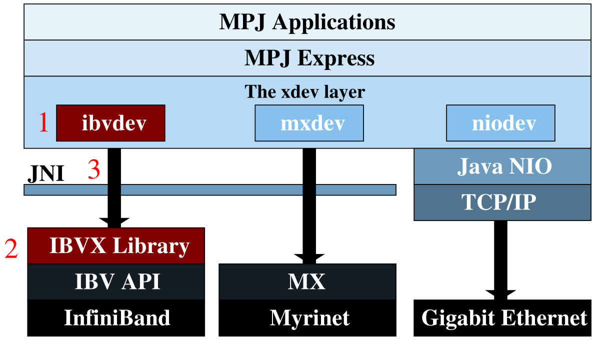
Listing 1 API of the xdev.Device class

4.3 Communication device design

Figure 4 presents the overall design of the communication middleware, which consists of three distinct parts. The first is the definition of a new device, `ibvdev`, in the `xdev` layer of MPJ Express (1 in Fig. 4). The analysis of the other high-speed network support in MPJ Express, the implementation of the `mxdev` device, reveals that it also uses native code via JNI to rely on the MX library, thus posing similar design issues as `ibvdev`. The MX library [23] provides a set of primitives similar to those needed to implement `xdev` interface, so there are a number of functions, such as `mx_isend`, `mx_issend`, `mx_irecv`, and `mx_wait`, that are used in the JNI layer. Therefore, `mxdev` acts as a Java wrapper layer to MX library, so that the implementation of a method in `xdev` generally delegates directly in a native method that performs the requested operation in MX library. Nevertheless, the design of `mxdev` is not directly applicable to `ibvdev` since InfiniBand lacks an MX-style library that implements the functionality and operations that must be implemented in `xdev`. The available communication layer for `ibvdev` is the IBV API, which offers low-level methods for the management of the HCA InfiniBand card.

Therefore, an MX-like library has been defined in order to provide `ibvdev` with a set of communication primitives with message-passing semantics on InfiniBand,

Fig. 4 Overall design of the communication library



to ease the development of the `xdev` communication device. This library has been denominated IBV eXpress (IBVX) (2 in Fig. 4). With this design, a native communication library has been implemented on top of IBV to provide basic message-passing communication primitives to higher level layers (either Java or non Java). Thus, the new communication device `ibvdev` can rely on IBVX through JNI. The design of this layer allows the access to IBVX from MPJ Express through its `ibvdev` device (3 in Fig. 4).

4.3.1 IBV eXpress library design

The IBVX library is a scalable and high performance low-level C message-passing middleware for communication on InfiniBand systems. It has been designed using the same approach as `xdev` communication devices. In fact, there is a mapping of `xdev` methods to IBVX functions, except for methods `id`, used for process identification, and `getSendOverhead` and `getRecvOverhead`, which are available only at the Java level as give information about the buffer handling. The IBVX API is presented in Listing 2. Like `xdev` API, IBVX includes only point-to-point communication, both blocking and nonblocking, and also synchronous communication support. In order to support nonblocking operations, IBVX implements `IBV_Wait` and `IBV_Test` functions, which handle nonblocking operation requests.

```
IBV_Init(char **pNames, int *pList, int nProcs, int rank, int psl);
IBV_Finalize();
IBV_Isend(void *buf, int size, int dst, int tag, int ctx, Request *r);
IBV_Issend(void *buf, int size, int dst, int tag, int ctx, Request *r);
IBV_Irecv(void *buf, int size, int src, int tag, int ctx, Request *r);
IBV_Send(void *buf, int size, int dst, int tag, int ctx);
IBV_Ssend(void *buf, int size, int dst, int tag, int ctx);
IBV_Recv(void *buf, int size, int src, int tag, int ctx, Status *s);
IBV_Wait(Request *request, Status *status);
IBV_Test(Request *request, Status *status);
IBV_Iprobe(int src, int tag, int context, Status *status);
IBV_Probe(int src, int tag, int context, Status *status);
Request* IBV_Peek();
```

Listing 2 Public interface of the IBV eXpress library

4.3.2 `ibvdev` *JNI layer design*

The design of the JNI layer of `ibvdev` is quite straightforward as it acts as a thin wrapper over IBVX. Thus, each native method of `ibvdev` delegates on a native IBVX function through JNI, implementing a series of three steps: (1) get Java objects associated parameters required for calling the corresponding library function in IBVX; (2) call IBVX function; and (3) save the results in the appropriate attributes of the Java objects involved in the communication. As general rules in the implementation of the JNI layer, it has been extensively used the caching of object references, thus minimizing the overhead associated with the JNI calls.

4.4 IBV eXpress library implementation

IBVX library implements nonblocking low-level communication primitives (see Listing 2) on top of IBV API. The first decision is the transport service used to create the queue pairs. Not all transports services support RDMA operations (see Table 1), whose support is desirable, so these transport services (UC and UD) are discarded.

Moreover, for RD and XRC transport services is not applicable the InfiniBand end-to-end flow control and this requires the development of a specific flow control software layer, which can add significant overhead if the implementation is not efficient. Therefore, the RC transport service has been selected as it provides reliability, delivery order, data loss detection, and error detection.

IBVX implements all communication operations as nonblocking communication primitives. Then blocking communication support is implemented as a nonblocking primitive followed by an `IBV_Wait` call. Therefore, the basic set of functions implemented consists of `IBV_Init`, `IBV_Finalize,` and nonblocking communication functions (`IBV_Isend`, `IBV_Issend`, `IBV_Irecv`), and the function that checks the completion of a nonblocking operation (`IBV_Test`). Thus, the operation that waits for the completion of a nonblocking operation (`IBV_Wait`) has been implemented following a strategy of polling (busy loop) as a continuous loop calling `IBV_Test` until the test is positive (thus minimizing latency). Blocking communication functions (`IBV_Send`, `IBV_Ssend`, and `IBV_Recv`) have been implemented by a call to its corresponding nonblocking function followed by an `IBV_Wait` call. Moreover, the probe operation, which checks for incoming messages without actual receipt of any of them, has been also implemented in the nonblocking version `IBV_Iprobe`, whereas the blocking version (`IBV_Probe`) relies on the nonblocking operation completion.

4.4.1 IBV eXpress communication protocols

Message-passing libraries usually implement two different communication protocols:

1. *Eager protocol*: the sender process eagerly sends the entire message to the receiver. In order to achieve this, the receiver needs to provide a sufficient number of buffers to handle incoming messages. This protocol has minimal startup over-

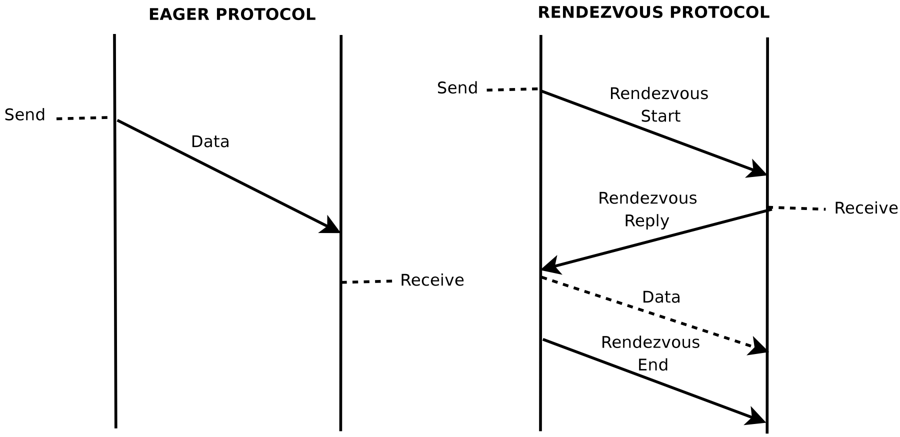
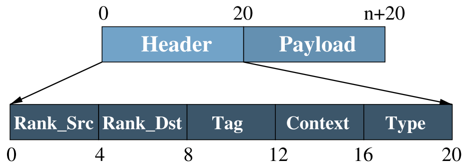


Fig. 5 MPI eager and rendezvous protocols

Fig. 6 Message format in IBVX library



heads and is used to implement low latency message-passing communication for smaller messages (typically $<$ 128 KB, configurable threshold).

2. *Rendezvous protocol*: this protocol negotiates (via control messages) the buffer availability at the receiver side before the message is actually transferred. This protocol is used for transferring large messages (typically $>$ 128 KB), whenever the sender is not sure whether the receiver actually has enough buffer space to hold the entire message.

Figure 5 presents graphically the operation of eager and rendezvous protocols.

4.4.2 Message format

The presence of control messages in the operation of the rendezvous protocol and the need for a receiving process to unequivocally distinguish a message, has forced the introduction of a message header before the actual data payload. Thus, a message is defined as the union of a header of 20 bytes (starting from the beginning), which is followed by the data payload, as shown in Fig. 6.

The header consists of 5 fields of 4 bytes each representing in this order: the process rank that sends the message, the destination process rank, the tag or label of the message, the context to which it belongs, and the type of message. All header fields are 4-byte integers, for all types of messages.

4.4.3 Eager protocol

The overhead of data copies is small for short messages, such as eager protocol transfers and control messages, which are eagerly push through the network to achieve the lowest latency. This operation matches with the semantic of InfiniBand send/receive communication.

In `IBV_Init` a reliable connection is set up between every two processes. For a single process, the send and receive queues of all connections are associated with a single CQ (Completion Queue). Through this CQ, the completion of all send and RDMA operations can be detected at the sender side. The completion of receive operations (or arrival of incoming messages) can also be detected through the CQ (see Fig. 7).

The InfiniBand Architecture requires the pinning of buffers previous to the communication, thus they must be registered with the hardware. In the eager protocol implementation (shown in Fig. 7), the buffer pinning and unpinning overhead is avoided by using a pool of prepinned, fixed size buffers for communication. For sending an eager data message, the data is copied to one of the buffers first and sent out from this buffer to the send queue (1 in Fig. 7). At the receiver side, a number of buffers from the pool are preposted (2 in Fig. 7). After the message is received, the payload is copied to the destination buffer (3 in Fig. 7). The communication of control messages also uses this buffer pool as they are actually sent using the eager protocol.

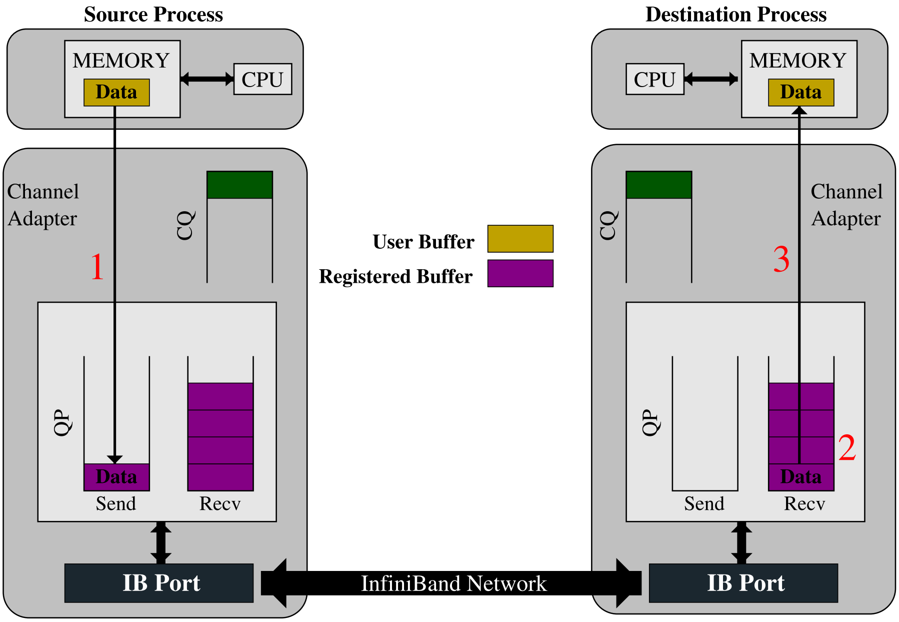


Fig. 7 Eager protocol implementation in IBVX

4.4.4 *Rendezvous protocol*

When transferring large messages it is extremely beneficial to avoid extra data copies. A zero-copy rendezvous protocol implementation can be achieved by using RDMA operations. The rendezvous protocol negotiates the buffer availability at the receiver side. However, the actual data can be transferred either by using RDMA Write or RDMA Read. RDMA Write-based approaches can totally eliminate intermediate copies and efficiently transfer large messages. RDMA Read-based approaches can enable both zero copy and computation and communication overlap. Similar approaches have been widely used for implementing MPI communications over different interconnects [24, 25].

The RDMA Write-based protocol is illustrated in Fig. 8 (right). In this implementation, the buffers are pinned down in memory and the buffer addresses are exchanged via control messages. The sending process first sends a control message to the receiver (RNDZ_START). The receiver replies to the sender using another control message (RNDZ_REPLY). This reply message contains the receiving application's buffer information along with the remote key to access that memory region. The sending process then sends the large message directly to the receiver's application buffer by using RDMA Write (DATA). Finally, the sending process issues another control message (RNDZ_END) which indicates to the receiver that the message has been placed in the application buffer.

IBVX uses a *progress engine* to discover incoming messages and to make progress on outstanding sends. As can be seen in Fig. 8, the RDMA Write-based rendezvous protocol generates multiple control messages which have to be discovered by the *progress engine*. Since the *progress engine* operation is based on polling, it requires a call to the IBVX library.

RDMA Read operation presents a small number of control messages and thus a reduced set of I/O bus transactions. In addition, since the receiver can progress

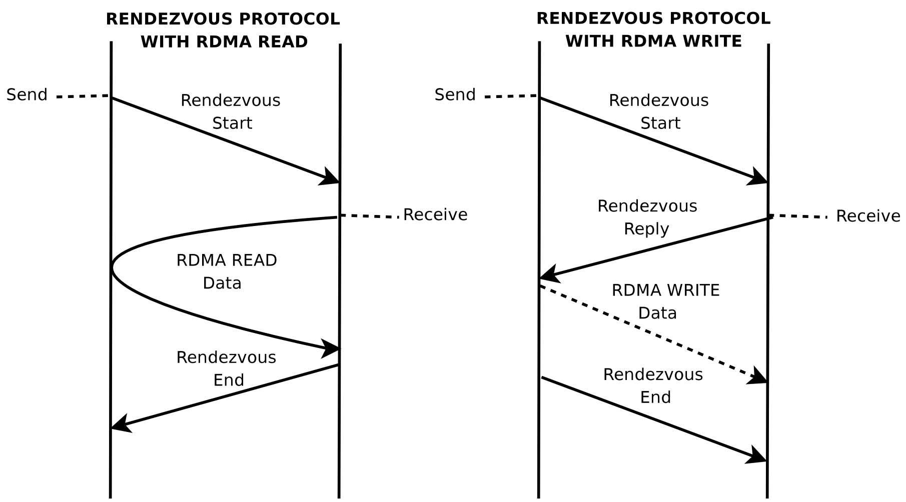


Fig. 8 Rendezvous protocol alternatives

independently of the sender (once the RNDZ_START message is sent), the sender does not need to call any IBVX progress, the data transfer proceeds with RDMA Read without direct control of the sender.

The rendezvous protocol over RDMA Read is also illustrated in Fig. 8 (left). Here, the sending process begins with the RNDZ_START message, which has embedded the virtual address and memory handle information of the message buffer to be sent. Thus, upon the receipt of this RNDZ_START message all the information about the application buffer is available to the receiving process, and no RNDZ_REPLY message needs to be sent any more. Upon its discovery, the receiving process issues the DATA message over RDMA Read. When the operation has been completed, it informs the sending process by a RNDZ_END message. This approach, although simple, poses several design challenges that have to be addressed before directly utilize RDMA Read:

- Limited Outstanding RDMA Reads: The number of outstanding RDMA Reads on any QP is a fixed number (typically 8 or 16), decided during the QP creation.
- Issuing RNDZ_END Message: According to InfiniBand specification [4], Send or RDMA Write transactions are not guaranteed to finish in order with outstanding RDMA Reads.

For these reasons, the rendezvous protocol has been implemented with RDMA Write operation, in order to benefit from a more productive development.

4.4.5 Cache of registered buffers

In rendezvous protocol, data buffers are pinned on-the-fly. However, the buffer pinning and unpinning overhead can be reduced by using the pin-down cache technique [26]. The idea is to maintain a cache of registered buffers. When a buffer is first registered, it is put into the cache. When the buffer is unregistered, the actual unregister operation is not carried out and the buffer stays in the cache. Thus, the next time when the buffer needs to be registered, we do not need to do anything because it is already in the cache. The effectiveness of pin-down cache depends on how often the application reuses its buffers. If the reuse rate is high, most of the buffer registration and deregistration operations can be avoided.

4.5 JNI layer implementation details

The JNI layer is a wrapper for IBVX library, in order to make it accessible from Java. Therefore, it implements the functions that the `javah` utility generated in terms of native operations contained in communication device Java classes. The development of this layer must take into account the design of the MPJ Express buffering layer [27]. The use of this buffering layer incurs a copying overhead that can be significant for large messages, and is considered a performance bottleneck for MPJ Express [28], so the handling of this layer has to be implemented efficiently.

The core class of the buffering layer used for packing and unpacking data is `mpjbuf.Buffer`. This class provides two storage options: static and dynamic. Implementations of static storage use the interface `mpjbuf.RawBuffer`. It is possible to have alternative implementations of the static section depending on the actual

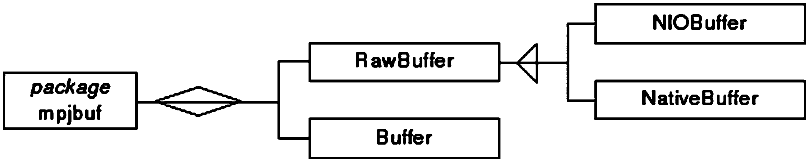


Fig. 9 Primary buffering classes in mpjbuf

raw storage medium. In addition, it also contains an attribute of type `byte[]` that represents the dynamic section of the message. Figure 9 shows two implementations of the `mpjbuf.RawBuffer` interface. The first, `mpjbuf.NIOBuffer` is an implementation based on `ByteBuffers`. The second, `mpjbuf.NativeBuffer` is an implementation for the native MPI device, which allocates memory in the native C code. Figure 9 shows the primary buffering classes in the `mpjbuf` API.

Regarding `mpjbuf.Buffer` class design, it is necessary to handle at the JNI layer a second call to the IBVX library when communicating a buffer with data in the two sections (static and dynamic). To support this operation efficiently, the first 4 bytes of the static buffer indicate the size of the dynamic part of the buffer. Thus, the overhead of this protocol in terms of buffering space, returned by `getSendOverhead` and `getRecvOverhead` methods, is 4 bytes. These methods, implemented for every MPJ Express communication device, are used to express the extra space needed in the static buffer to implement the buffering layer support, and they are profusely used when handling the buffer contents.

5 Performance evaluation

This section presents a performance evaluation of the developed communication device `ibvdev`, compared to native MPI libraries (MVAPICH and OpenMPI) and the MPJ Express communications devices `niodev` over InfiniBand (using IPoIB) and `smpdev` for shared memory communication. This evaluation consists of a microbenchmarking of point-to-point data transfers (Sect. 5.2) and collective communications (Sect. 5.3), as well as an analysis of the impact on the overall performance of the use of the developed library on several representative MPJ codes (Sect. 5.4).

5.1 Experimental configuration

The evaluation of `ibvdev` has been carried out in a cluster which consists of 8 nodes, each of them with 8 GB of RAM and 2 Intel Xeon E5520 quad-core Nehalem processors. Although each node has 8 cores, the HyperThreading (HT) is enabled so it is possible to run 16 processes per node concurrently. The interconnection networks are InfiniBand (16 Gbps of maximum theoretical bandwidth), with OFED driver 1.5, and Gigabit Ethernet (1 Gbps). The OS is Linux CentOS 5.3 with kernel 2.6.18 and the JVM is Sun JDK 1.6.0_13. The evaluated MPJ implementation is MPJ Express [29] version 0.36 (labeled MPJE in graphs) and the evaluated MPI implementations are MVAPICH [25] v1.2.0 and OpenMPI [24] v1.3.3. The PSL (Protocol Switch Limit)

MPJ Express attribute, the threshold between eager and rendezvous send protocols, has been set to 128 KB message size for all the benchmarks. F-MPJ and MPJ/Ibis results are not shown for clarity purposes, apart from the fact that `ibvdev` is only integrated in MPJ Express. However, as they are sockets-based implementations, their performance is similar to `niodev` results.

5.2 Point-to-point micro-benchmarking

In order to micro-benchmark MPJ point-to-point and collectives primitives performance our own micro-benchmark suite [30], similar to Intel MPI Benchmarks used for MPI libraries, has been used due to the lack of suitable micro-benchmarks for MPJ evaluation. Here, the results shown are the half of the round-trip time of a pingpong test or its corresponding bandwidth. The transferred data are byte arrays, avoiding the serialization overhead that would distort the analysis of the results.

Figures 10 and 11 show point-to-point latencies (for short messages) and bandwidths (for long messages) on InfiniBand and shared memory, respectively. The `ibvdev` middleware obtains significant point-to-point performance benefits, thus obtaining 11 µs start-up latency and up to 7.2 Gbps bandwidth. The threshold between eager and rendezvous send protocols can be observed in the bandwidth graph at 128 KB, which confirms the efficiency of the implementation of the zero-copy rendezvous protocol with RDMA Write for `ibvdev`. These results outperform significantly `niodev` over InfiniBand, limited to 65 µs start-up latency and below 3 Gbps bandwidth.

Compared to native MPI libraries, `ibvdev` obtains a similar bandwidth than MVAPICH (7 Gbps) in this testbed, surpassing it even at several points (e.g., 32 KB, 256 KB, and 512 KB message sizes). Nevertheless, OpenMPI shows the best performance from 32 KB message size, obtaining up to 9.2 Gbps bandwidth. As for latency, `ibvdev` obtains better results than MVAPICH (13 µs) and only slightly worse than OpenMPI (10 µs), again the best performer.

Regarding shared memory communication performance, `ibvdev` obtains much better start-up latency, 6 µs, than the multithreading `smpdev` middleware, which achieves 17 µs, which means that `ibvdev` has implemented a highly efficient communication protocol and that `smpdev` presents poor start-up latency, caused by an excess of synchronizations. The native MPI libraries are again the best performers obtaining 0.5 µs an 1 µs for MVAPICH and OpenMPI, respectively, due to their efficient communications support on shared memory. Regarding bandwidth, MPJ devices are far from native MPI libraries, obtaining worse performance (15.3 Gbps and 22 Gbps for `ibvdev` and `smpdev`, respectively).

5.3 Collective primitives micro-benchmarking

Figure 12 presents the aggregated bandwidth for representative MPJ data movement operations (broadcast and allgather), and computational operations (reduce and allreduce double precision sum operations) with 128 processes. The aggregated bandwidth metric has been selected as it takes into account the global amount of data transferred. The `niodev` allgather results could not be taken due to flaws in the implementation that hanged its operation. In addition to `ibvdev`, `niodev`, and MPI

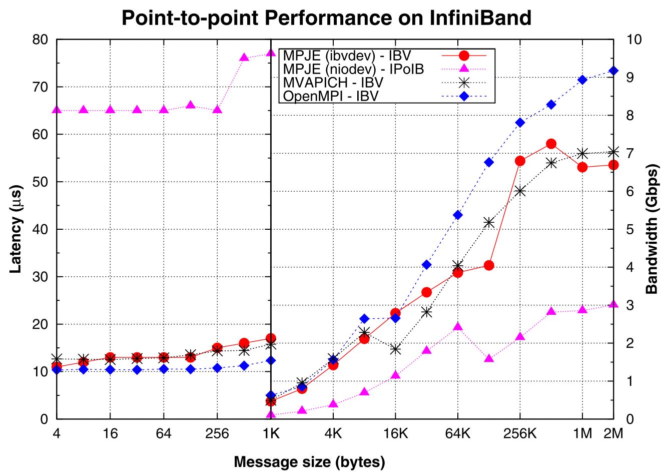


Fig. 10 Message-passing point-to-point performance on InfiniBand

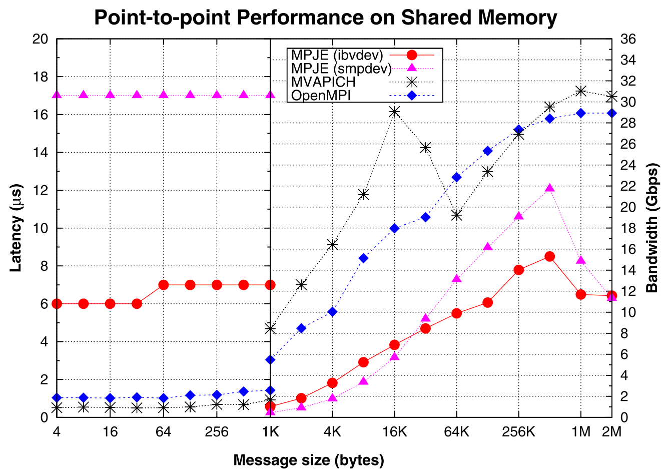


Fig. 11 Message-passing point-to-point performance on shared memory

communications it has been evaluated the performance of multithreaded versions of the MPJ collective operations, running only one process per node, and 16 threads within each process. Thus, instead of running 128 processes on the cluster, only 8 processes are being used, taking advantage of intranode communications through multithreading. This hybrid support of network and multithreading communications is one of the main advantages of Java middleware for scalable and efficient communication on clusters of multicore processors.

The results confirm that `ibvdev` outperforms significantly `niodev`, achieving up to one order of magnitude higher performance, although generally the performance benefit is 2 or 3 times better. Moreover, both `ibvdev` and `niodev` take advantage of the multithreaded collectives. With respect to the MPI libraries, `ibvdev`

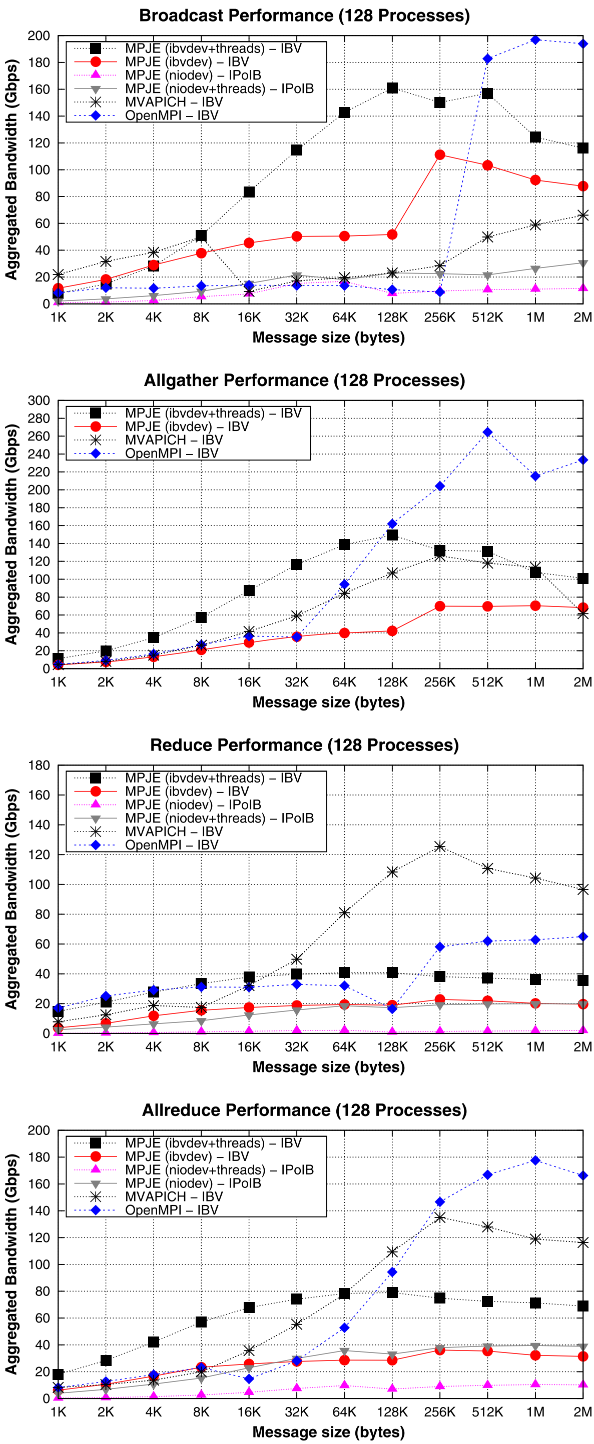


Fig. 12 Message-passing collective primitives performance

achieves better performance than MPI collectives for short messages, up to 16–256 KB, thanks to the exploitation of multithreading in collectives implementation and the use of a high PSL (128 KB), whereas MPI libraries use smaller PSL (8 KB). However, for longer messages the MPI collectives achieve much better performance due to the use of better collective algorithms, and the use of pipelined transfers.

5.4 Kernel/application performance analysis

The impact of `ibvdev` on the scalability of Java parallel codes has been analyzed using the NAS Parallel Benchmarks (NPB) implementation for MPJ (NPB-MPJ) [31], selected as the NPB are probably the benchmarks most commonly used in the evaluation of languages, libraries, and middleware for HPC. In fact, there are implementations of the NPB for MPI, OpenMP, and hybrid MPI/OpenMP.

Four representative NPB codes have been evaluated: CG (Conjugate Gradient), FT (Fourier Transform), IS (Integer Sort), and MG (Multi-Grid). Moreover, the jGadget [32] cosmology simulation application has also been analyzed. These MPJ codes have been selected as they show very poor scalability with MPJ Express over InfiniBand. Hence, these are target codes for the evaluation of the impact on performance of the use of `ibvdev` in MPJ Express. The results have been obtained using up to 64 processes instead of 128, due to memory constraints on the cluster.

Figure 13 shows the NPB-MPJ CG, IS, FT, and MG results, respectively, for the Class C workload in terms of MOPS (Millions of Operations Per Second) (left) and its corresponding scalability, in terms of speedup (right). For CG kernel, `ibvdev` doubles the performance of the `niodev` device over InfiniBand, with almost 9000 MOPS compared to less than 4000 MOPS on 64 processes. With respect to IS kernel, the results for `niodev` over InfiniBand show a significant slowdown with 64 processes, not taking advantage of the use of 64 processes, while `ibvdev` keeps on scaling and gets up to 650 MOPS, significantly outperforming the `niodev` results. Regarding FT, `ibvdev` also doubles the performance of the `niodev` device over InfiniBand, with around 17000 MOPS compared to less than 8000 MOPS. Finally, the impact of `ibvdev` on MG is smaller than for the remaining codes as this NPB is less communication intensive, as obtains relatively good speedups, even with `niodev` (speedup of 30 with 64 processes).

The performance comparison of `ibvdev` against MPI libraries has two different analyses, depending on the metric used. If we take into account the MOPS achieved, MPI benchmarks obtain always the best performance, around a 50% higher than `ibvdev` results. The poorer performance of NPB-MPJ can be attributed to the lower performance of the JVM compared to native compilers. However, if we have a look at the speedups, `ibvdev` outperforms MPI for FT and MG, while obtains slightly lower scalability for CG and IS, which suggests that `ibvdev` implements a highly efficient communication support, even comparable to MPI libraries, and that the use of efficient communication libraries can bridge the gap between Java and natively compiled languages provided that an efficient communication support is made available.

The jGadget application is the MPJ implementation of Gadget [33], a popular cosmology simulation code initially implemented in C and parallelized using MPI that is used to study a large variety of problems like colliding and merging galaxies or the

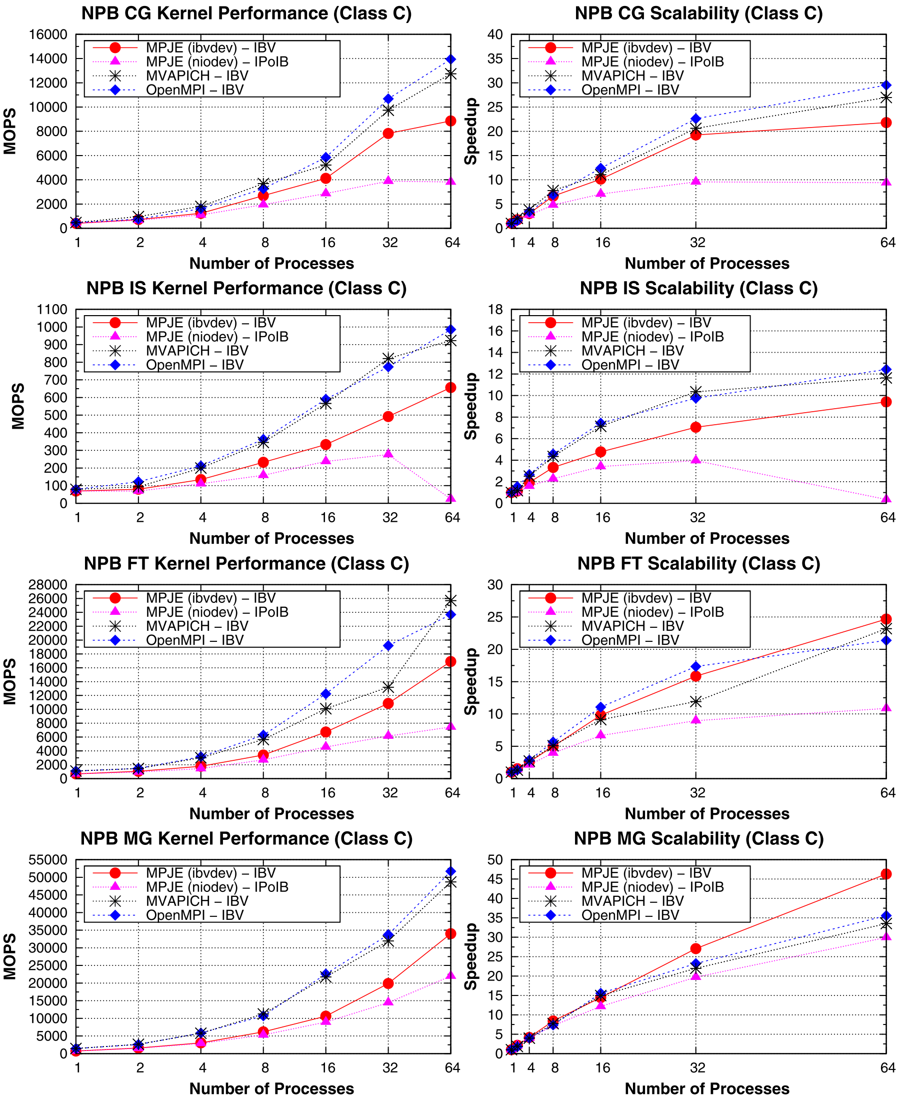
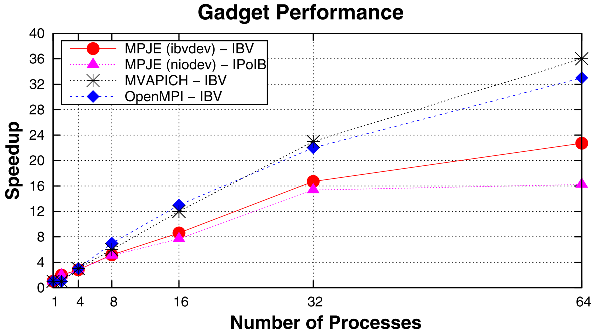


Fig. 13 Performance and scalability of NPB-MPI/MPJ codes

formation of large-scale structures. This application has been selected for the performance evaluation of ibvdev, measuring its performance using up to 64 processes instead of 128, due to memory constraints on the cluster (each Java process is using its own JVM).

Figure 14 presents the performance results of jGadget running a two million particles cluster formation simulation. As jGadget is a communication-intensive application, with important collective operations overhead, only modest speedups are obtained. Here, ibvdev can take advantage of the use of 64 processes (speedup above 22), whereas niodev over IPoIB remains with a speedup of 16. Regarding

Fig. 14 Scalability of MPI/MPJ Gadget



MPI results, OpenMPI and MVAPICH achieve around 45% higher speedup than `ibvdev` on 64 processes, which suggests that this middleware is bridging the gap between Java and natively compiled applications in HPC.

6 Conclusions

This paper has presented `ibvdev`, a scalable and efficient low-level Java message-passing device for communication on InfiniBand systems. The increase in the number of cores per system demands languages with built-in multithreading and networking support, such as Java, as well as scalable and efficient communication middleware that can take advantage of multicore systems. The developed device transparently provides Java message-passing applications with efficient performance on InfiniBand thanks to its direct support on IBV and the efficient and scalable implementation of a lightweight communication protocol which is able to take advantage of RDMA over InfiniBand.

The performance evaluation of `ibvdev` on an InfiniBand multicore cluster has shown that this middleware obtains start-up latencies and bandwidths similar to MPI performance results, obtaining in fact up to 85% start-up latency reduction and twice the bandwidth compared to previous Java middleware on InfiniBand. Additionally, the impact of `ibvdev` on message-passing collective operations is significant, achieving up to one order of magnitude performance increases compared to previous Java solutions, especially when taking advantage of shared memory intra-process (multithreading) communication. The analysis of the impact of the use of `ibvdev` on MPJ applications shows a significant performance increase compared to sockets-based middleware (`niodev`), which helps to bridge the gap between Java and natively compiled codes in HPC. To sum up, the efficiency of this middleware, which is even competitive with MPI point-to-point transfers, increments the scalability of communications intensive Java applications, especially in combination with the native multithreading support of Java.

Acknowledgements This work was funded by the Ministry of Science and Innovation of Spain under Project TIN2010-16735, and by the Xunta de Galicia under the Consolidation Program of Competitive Research Groups and Galician Network of High Performance Computing.

References

1. Taboada GL, Ramos S, Expósito RR, Touriño J, Doallo R (2011, in press) Java in the high performance computing arena: research, practice and experience. Sci Comput Program. doi:10.1016/j.scico.2011.06.002
2. Blount B, Chatterjee S (1999) An evaluation of Java for numerical computing. Sci Program 7(2):97–110
3. Shafi A, Carpenter B, Baker M (2009) Nested parallelism for multi-core HPC systems using Java. J Parallel Distrib Comput 69(6):532–545
4. InfiniBand Trade Association (2004) Infiniband architecture specification volume 1, release 1.2.1. http://www.infinibandta.org/ [Last visited: April 2011]
5. Baker M, Carpenter B, Shafi A (2005) A pluggable architecture for high-performance Java messaging. IEEE Distrib Syst Online 6(10):1–4
6. IETF Draft. IP over IB. http://www.ietf.org/old/2009/ids.by.wg/ipoib.html [Last visited: April 2011]
7. Hongwei Z, Wan H, Jizhong H, Jin H, Lisheng Z (2007) A performance study of Java communication stacks over InfiniBand and Gigabit Ethernet. In: Proc 4th IFIP intl conf network and parallel computing (NPC'07), Dalian, China, pp 602–607
8. Veldema R, Hofman RFH, Bhoedjang R, Bal HE (2003) Run-time optimizations for a Java DSM implementation. Concurr Comput 15(3–5):299–316
9. Righi RDR, Navaux POA, Cera MC, Pasin M (2005) Asynchronous communication in Java over InfiniBand and DECK. In: Proc 17th intl symp on computer architecture and high performance computing (SBAC-PAD'05), Rio de Janeiro, Brazil, pp 176–183
10. Taboada GL, Touriño J, Doallo R (2008) Java Fast Sockets: enabling high-speed Java communications on high performance clusters. Comput Commun 31(17):4049–4059
11. Wan H, Hongwei Z, Jin H, Jizhong H, Lisheng Z (2007) Jdib: Java applications interface to unshackle the communication capabilities of InfiniBand networks. In: Proc 4th IFIP intl conf network and parallel computing (NPC'07), Dalian, China, pp 596–601
12. Wan H, Jizhong H, Jin H, Lisheng Z, Yao L (2008) Enabling RDMA capability of InfiniBand network for Java applications. In: Proc 4th IFIP intl conf on networking, architecture, and storage (NAS'08), Chongqing, China, pp 187–188
13. Yao L, Jizhong H, Jinjun G, Xubin H (2009) uStream: a user-level stream protocol over InfiniBand. In: Proc 15th intl conf on parallel and distributed systems (ICPADS'09), Shenzhen, China, pp 65–71
14. Dunning D, Regnier G, McAlpine G, Cameron D, Shubert B, Berry F, Merritt AM, Gronke E, Dodd C (1998) The Virtual Interface Architecture. IEEE MICRO 18(2):66–76
15. Compaq, Intel and Microsoft Corporations (1997) The Virtual Interface Architecture specification, version 1.0
16. OpenFabrics Alliance Website. http://www.openfabrics.org/ [Last visited: April 2011]
17. Carpenter B, Fox G, Ko S-H, Lim S mpiJava 1.2: API specification. http://www.hpjava.org/reports/mpiJava-spec/mpiJava-spec/mpiJava-spec.html [Last visited: April 2011]
18. The mpiJava project. http://www.hpjava.org/mpiJava.html [Last visited: April 2011]
19. Bornemann M, van Nieuwpoort RV, Kielmann T (2005) MPJ/Ibis: a flexible and efficient message passing platform for Java. In: Proc. 12th European PVM/MPI users' group meeting (EuroPVM/MPI'05), Sorrento, Italy, pp 217–224
20. Taboada GL, Touriño J, Doallo R (2011) F-MPJ: scalable Java message-passing communications on parallel systems. J Supercomput. doi:10.1007/s11227-009-0270-0
21. Java Grande Forum. http://www.javagrande.org [Last visited: April 2011]
22. van Nieuwpoort RV, Maassen J, Wrzesinska G, Hofman R, Jacobs C, Kielmann T, Bal HE (2005) Ibis: a flexible and efficient Java-based grid programming environment. Concurr Comput 17(7–8):1079–1107
23. MX User's Guide. http://www.myri.com/scs/MX/doc/mx.pdf [Last visited: April 2011]
24. Open Source High Performance MPI Library. http://www.open-mpi.org/ [Last visited: April 2011]
25. MPI over InfiniBand, 10GigE/iWARP and RDMAoE. http://mvapich.cse.ohio-state.edu/ [Last visited: April 2011]
26. Tezuka H, O'Carroll F, Hori A, Ishikaw Y (1998) Pin-down cache: a virtual memory management technique for zero-copy communication. In: Proc 12th intl parallel processing symposium/9th symposium on parallel and distributed processing (IPPS/SPDP'98), Orlando, FL, USA, pp 308–314
27. Baker M, Carpenter B, Shafi A (2007) A buffering layer to support derived types and proprietary networks for Java HPC. Scalable Comput Pract Experience 8(4):343–358

28. Taboada GL, Touriño J, Doallo R, Shafi A, Baker M, Carpenter B (2011, in press) Device level communication libraries for high-performance computing in Java. Concurr Comput. doi:110.1002/cpe.1777
29. MPJ Express Website. http://mpj-express.org/ [Last visited: April 2011]
30. Taboada GL, Touriño J, Doallo R (2003) Performance analysis of Java message-passing libraries on Fast Ethernet, Myrinet and SCI clusters. In: Proc 5th IEEE intl conf on cluster computing (CLUSTER'03), Hong Kong, China, pp 118–126
31. Mallón DA, Taboada GL, Touriño J, Doallo R (2009) NPB-MPJ: NAS parallel benchmarks implementation for message-passing in Java. In: Proc 17th Euromicro intl conf on parallel, distributed, and network-based processing (PDP'09), Weimar, Germany, pp 181–190
32. Baker M, Carpenter B, Shafi A (2006) MPJ Express meets Gadget: towards a Java code for cosmological simulations. In: 13th European PVM/MPI users' group meeting (EuroPVM/MPI'06), Bonn, Germany, pp 358–365
33. Springel V (2005) The cosmological simulation code GADGET-2. Mon Not R Astron Soc 364(4):1105–1134