CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015; **27**:4852–4879
Published online 15 May 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3473

Low-latency Java communication devices on RDMA-enabled networks

Roberto R. Expósito*,†, Guillermo L. Taboada, Sabela Ramos, Juan Touriño and Ramón Doallo

Computer Architecture Group, Department of Electronics and Systems, University of A Coruña, A Coruña, Spain

SUMMARY

Providing high-performance inter-node communication is a key capability for running high performance computing applications efficiently on parallel architectures. In fact, current systems deployments are aggregating a significant number of cores interconnected via advanced networking hardware with Remote Direct Memory Access (RDMA) mechanisms, that enable zero-copy and kernel-bypass features. The use of Java for parallel programming is becoming more promising thanks to some useful characteristics of this language, particularly its built-in multithreading support, portability, easy-to-learn properties, and high productivity, along with the continuous increase in the performance of the Java virtual machine. However, current parallel Java applications generally suffer from inefficient communication middleware, mainly based on protocols with high communication overhead that do not take full advantage of RDMA-enabled networks. This paper presents efficient low-level Java communication devices that overcome these constraints by fully exploiting the underlying RDMA hardware, providing low-latency and high-bandwidth communications for parallel Java applications. The performance evaluation conducted on representative RDMA networks and parallel systems has shown significant point-to-point performance increases compared with previous Java communication middleware, allowing to obtain up to 40% improvement in application-level performance on 4096 cores of a Cray XE6 supercomputer. Copyright © 2015 John Wiley & Sons, Ltd.

Received 30 April 2014; Revised 28 November 2014; Accepted 18 January 2015

KEY WORDS: parallel systems; Remote Direct Memory Access (RDMA); RDMA-enabled networks; Java communication middleware; Message-Passing in Java (MPJ)

1. INTRODUCTION

Java is a highly portable and flexible programming language, enjoying a dominant position in a wide diversity of computing environments. Some of the interesting features of Java are its built-in multithreading support in the core of the language, object orientation, automatic memory management, type-safety, platform independence, portability, easy-to-learn properties, and thus, higher productivity. Furthermore, Java has become the leading programming language both in academia and industry.

The Java Virtual Machine (JVM) is currently equipped with efficient Just-in-Time (JIT) compilers that can obtain near-native performance from the platform-independent bytecode [1]. In fact, the JVM identifies sections of the code frequently executed and converts them to native machine code instead of interpreting the bytecode. This significant improvement in its computational performance has narrowed the performance gap between Java and natively compiled languages (e.g., C/C++, Fortran). Thus, Java is currently gaining popularity in other domains, which usually make use of High Performance Computing (HPC) infrastructures, such as the area of parallel computing [2, 3] or

*Correspondence to: Roberto R. Expósito, Computer Architecture Group, Department of Electronics and Systems, University of A Coruña, Spain.
†E-mail: rreye@udc.es

in Big Data analytics, where the Java-based Hadoop distributed computing framework [4] is among the preferred choices for the development of applications that follow the MapReduce programming model [5].

With the continuously increasing number of cores in current HPC systems to meet the ever growing computational power needs, it is vitally important for communication middleware to provide efficient inter-node communications on top of high-performance interconnects. Modern networking hardware provides Remote Direct Memory Access (RDMA) capabilities that enable zero-copy and kernel-bypass features, key mechanisms for obtaining scalable application performance. However, it is usually difficult to program directly with RDMA hardware. In this context, it is fundamental to fully harness the power of the likely abundant processing resources and take advantage of the interesting features of RDMA networks with still ease-to-use programming models. The Message-Passing Interface (MPI) [6] remains as the de-facto standard in the area of parallel computing, being the most commonly used programming model for writing C/C++ and Fortran parallel applications, but remains out of the scope of Java. The main reason is that current parallel Java applications usually suffer from inefficient communication middleware, mainly based on protocols with high overhead that do not take full advantage of RDMA-enabled networks [7]. The lack of efficient RDMA hardware support in current Message-Passing in Java (MPJ) [8] implementations usually results in lower performance than natively compiled codes, which has prevented the use of Java in this area. Thus, the adoption of Java as a mainstream language on these systems heavily depends on the availability of efficient communication middleware in order to benefit from its appealing features at a reasonable overhead.

This paper focuses on providing efficient low-level communication devices that overcome these constraints by fully exploiting the underlying RDMA hardware, enabling low-latency and high-bandwidth communications for Java message-passing applications. The performance evaluation conducted on representative RDMA networks and parallel systems has shown significant point-to-point performance improvements compared with previous Java message-passing middleware, in addition to higher scalability for communication-intensive HPC codes. These communication devices have been integrated seamlessly in the FastMPJ middleware [9], our Java message-passing implementation, in order to make them available for current MPJ applications. Therefore, this paper presents our research results on improving the RDMA network support in FastMPJ, which would definitely contribute to increase the use of Java in parallel computing. More specifically, the main contributions of this paper are the following:

- The design and implementation of two new low-level communication devices, `ugnidev` and `mxmdev`. The former device is intended to provide efficient support for the RDMA networks used by the Cray XE/XK/XC family of supercomputers. The latter includes support for the recently released messaging library developed by Mellanox for its RDMA adapters.
- An enhanced version of the `ibvdev` communication device for InfiniBand systems [10], which now includes new support for RDMA networks along with an optimized communication protocol to improve short-message performance.
- An experimental comparison of representative MPJ middleware, which includes a micro-benchmarking of point-to-point primitives on several RDMA networks and an application-level performance analysis conducted on two parallel systems: a multi-core InfiniBand cluster and a large Cray XE6 supercomputer.

The remainder of this paper is organized as follows. Section 2 presents background information about RDMA networks and their software support. Section 3 introduces the related work. Section 4 presents the overall design of `xxdev`, the low-level communication device layer included in FastMPJ. This is followed by Sections 5, 6, and 7, which describe the design and implementation of the new `xxdev` communication devices presented in this paper: `ugnidev`, `ibvdev`, and `mxmdev`, respectively. Section 8 shows the performance results of the developed devices gathered from a micro-benchmarking of point-to-point primitives on several RDMA networks. Next, this section analyzes the impact of their use on the overall performance of representative Java HPC codes. Finally, our concluding remarks are summarized in Section 9.

DOI: 10.1002/cpe

2. OVERVIEW OF RDMA-ENABLED NETWORKS

Most high-performance clusters and custom supercomputers are deployed with high-speed interconnects. These networking technologies typically rely on scalable topologies and advanced network adapters that provide RDMA-capable specialized hardware to enable zero-copy and kernel-bypass facilities. Some of the main benefits of using RDMA hardware are low-latency and high-bandwidth inter-node communication with low CPU overhead.

In recent years, the InfiniBand (IB) architecture [11] has become the most widely adopted RDMA networking technology in the TOP500 list [12], especially for multi-core clusters. In addition, two other popular RDMA implementations, the Internet Wide Area RDMA Protocol (iWARP) [13] and RDMA over Converged Ethernet (RoCE) [14], have also been proposed to extend the advantages of RDMA technologies to ubiquitous Internet Protocol (IP)/Ethernet-based networks. On the one hand, iWARP defines how to perform RDMA over a connection-oriented transport such as the Transmission Control Protocol (TCP). Thus, iWARP includes a TCP Offload Engine (TOE) to offload the whole TCP/IP stack onto the hardware, while the Direct Data Placement protocol [15] implements the zero-copy and kernel-bypass mechanisms. On the other hand, RoCE takes advantage of the more recent enhancements to the Ethernet link layer. The IEEE Converged Enhanced Ethernet is a set of standards, defined by the Data Center Bridging (DCB) task group [16] within IEEE 802.1, which are intended to make Ethernet reliable and lossless (like IB). This allows the IB transport protocol to be layered directly over the Ethernet link layer. Hence, RoCE utilizes the same transport and network layers from the IB stack and swaps the link layer for Ethernet, providing IB-like performance and efficiency to ubiquitous Ethernet infrastructures. Compared with iWARP, RoCE is a more natural extension of message-based transfers and, therefore, usually offers better efficiency than iWARP. However, one disadvantage of RoCE is that it requires DCB-compliant Ethernet switches, as it does not operate with standard ones.

Although the current market is dominated by clusters, many of the most powerful computing installations are custom supercomputers [12] that usually rely on specifically designed Operating Systems (OS) and proprietary RDMA-enabled interconnects. Some examples are the IBM Blue Gene/Q (BG/Q) and the Cray XE/XK/XC family of supercomputers. On the one hand, the compute nodes of the IBM BG/Q line are interconnected via a custom 5D torus network [17]. On the other hand, Cray XE/XK architectures include the Gemini interconnect [18] based on a 3D torus topology, while the XC systems provide the Aries interconnect that uses a novel network topology called Dragonfly [19].

2.1. Software support

The IB architecture has no standard Application Programming Interface (API) within the specification. It only defines the functionality provided by the RDMA adapter in terms of an abstract and low-level interface called Verbs[‡], which has initially resulted in different vendors developing their own incompatible APIs. For instance, one of the first proprietary interfaces available for IB was the Mellanox Verbs API (mVAPI). However, mVAPI is vendor-specific and IB-specific (i.e., it cannot work either with non-Mellanox hardware or iWARP adapters), and it is currently deprecated. The *de facto* standard is the implementation of the Verbs interface developed by the OpenFabrics Alliance (OFA) [20], which includes both user-level and kernel-level APIs. This open-source software stack has been adopted by most vendors, and it is released as part of the OpenFabrics Enterprise Distribution (OFED). As a software stack, OFED spans both the OS kernel, providing hardware-specific drivers, and the user space, implementing the Verbs interface. Although OFED was initially developed to work over IB networks, currently, it also includes support for iWARP and RoCE. Hence, it offers a uniform and transport-independent low-level API for the development of RDMA and kernel-bypass applications on IB, iWARP, and RoCE interconnects. In addition to the OFED stack, some vendors provide additional user-space libraries that are specifically designed for their RDMA hardware. Examples of these libraries are the Performance Scaled Messaging (PSM)

[‡]A *verb* is a semantic description of a function that must be provided.

DOI: 10.1002/cpe

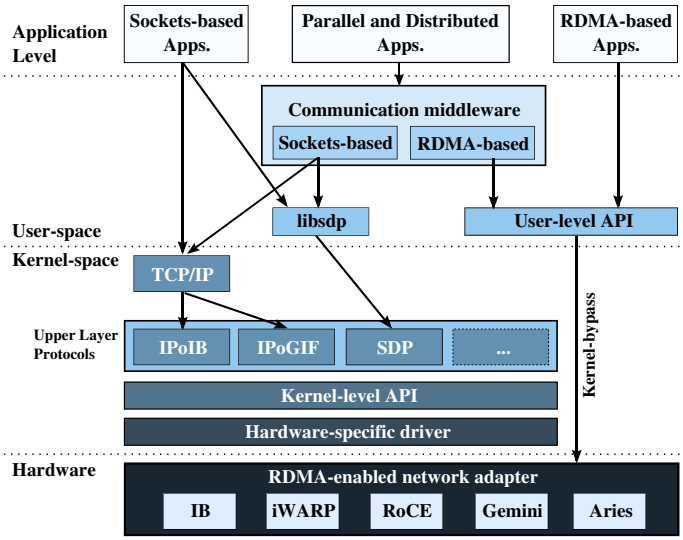


Figure 1. Overview of the RDMA software stack.

and MellanoX Messaging (MXM), which are currently available for Intel/QLogic and Mellanox adapters, respectively. These libraries can offer a higher level API than Verbs, usually also matching some of the needs of upper-level communication middleware (e.g., message-passing libraries). Regarding supercomputer systems, vendors provide a specific interface to their custom interconnects intended to be used for user-space communication. These interfaces are usually low-level APIs that directly expose the RDMA capabilities of the hardware (like Verbs), on top of which the communication middleware and applications can be implemented. For instance, IBM includes the System's Programming Interface (SPI) to program the torus-based interconnect of the BG/Q system, while Cray provides two different interfaces for implementing communication libraries targeted for Gemini/Aries interconnects: Generic Network Interface (GNI) and Distributed Memory Application (DMAPP). Note that all these programming interfaces are only available in C, and therefore, any communication support from Java must resort to the Java Native Interface (JNI).

Finally, existing sockets-based middleware and applications are usually able to run over RDMA networks without rewriting, using additional extensions known as Upper Layer Protocols (ULP). Examples of ULPs are the IP emulation over IB (IPoIB) [21] and the IP over Gemini Fabric (IPoGIF) modules. However, these ULPs are unable to take full advantage of the RDMA hardware, introducing additional TCP/IP processing overhead and performance penalties (e.g., multiple data copies and high CPU utilization) compared with native RDMA interfaces. In order to overcome these issues, some high-performance sockets implementations are available as additional ULPs. For instance, the Sockets Direct Protocol (SDP) [22] provides a user-space preloadable library and kernel module that bypasses the TCP/IP stack to take advantage of the IB/iWARP/RoCE hardware features. However, SDP has limited utility as only applications relying on the TCP/IP sockets API can use it, and other IP stack uses or TCP layer modifications (e.g., IPSec and UDP) cannot benefit from it. In addition, because of the restrictions of the socket interface, SDP cannot provide the low latencies of native RDMA. Furthermore, OpenFabrics has recently ended the support for SDP and now is considered deprecated. Figure 1 provides a graphical overview of the described RDMA software support.

3. RELATED WORK

There have been several earlis our high-performance Java socketsy works about Java for HPC soon after its release that have identified its potential for scientific computing [23, 24]. Moreover, some projects have been focused particularly on Java communication efficiency. These related works can be classified into the following: (1) Java over the Virtual Interface Architecture (VIA) [25]; (2) Java sockets implementations; (3) Java Remote Method Invocation (RMI) protocol optimizations; (4)

DOI: 10.1002/cpe

Java Distributed Shared Memory (DSM) projects; (5) low-level Java libraries on RDMA networks; and (6) efficient MPJ middleware.

Javia [26] and Jaguar [27] provide access to high-speed cluster networks through VIA. The VIA architecture is one of the several approaches for user-level networking developed in the 1990s, which has served as basis for IB. More specifically, Javia reduces data copying using native buffers, and Jaguar acts as a replacement of the JNI layer in the JVM, providing an API to access VIA. Their main drawbacks are the use of particular APIs, the need for modified Java compilers that ties the implementation to a certain JVM, and the lack of non-VIA communication support. Additionally, Javia exposes programmers to buffer management and uses a specific garbage collector.

The widespread socket API can be considered as the standard low-level communication layer. Thus, sockets have been the choice for implementing in Java the lowest level of network communication. However, Java sockets lack efficient high-speed network support and HPC tailoring, so they have to resort to inefficient TCP/IP emulations (e.g., IPoIB) for full networking support [7]. Ibis sockets partly solve these issues adding Myrinet support and being the base of Ibis [28], a parallel and distributed Java computing framework. However, Ibis lacks support for current RDMA networks, and its implementation on top of JVM sockets limits the performance benefits to serialization improvements. Aldeia [29] is a proposal of an asynchronous sockets communication layer over IB whose preliminary results were encouraging, but requires an extra copy to provide asynchronous write operations, which incurs an important overhead, whereas the read method is synchronous. Java Fast Sockets (JFS) [30] is our high-performance Java sockets implementation that relies on SDP (see Figure 1) to support Java communications over IB. JFS avoids the need for primitive data type array serialization and reduces buffering and unnecessary copies. Nevertheless, the use of the socket API still represents an important source of overhead and lack of scalability in Java communications, especially in the presence of high-speed networks [7].

Other related work about performance optimization of Java communications included many efforts in RMI, which is a common communication facility for Java applications. ProActive [31] is a fully portable 'pure' Java (i.e., 100% Java) RMI-based middleware for parallel, multithreaded and distributed computing. Nevertheless, the use of RMI as its default transport layer adds significant overhead to the operation of this middleware. Therefore, the optimization of the RMI protocol has been the goal of several projects, such as KaRMI [32], Manta [33], Ibis RMI [28], and Opt RMI [34]. However, the use of non-standard APIs, the lack of portability and the insufficient overhead reductions, still significantly larger than socket latencies, have restricted their applicability. Therefore, although Java communication middleware used to be based on RMI, current middleware use sockets due to their lower overhead.

Java DSM projects are usually based on sockets and thus benefit from socket optimizations, but their performance on top of high-speed networks still suffers from significant communication overheads. In order to reduce their impact, two DSM projects have implemented their communications relying on low-level libraries: CoJVM [35] uses VIA, whereas Jackal [36] includes RDMA support through the Verbs API [37]. Nevertheless, these projects share unsuitable characteristics such as the use of modified JVMs, the need of source code modification and limited interoperability and portability (e.g., Jackal is a Java-to-native compiler that does not provide any API to Java developers, implementing data transfers specifically for Jackal).

Other approaches are low-level Java libraries restricted to specific RDMA networks. For instance, Jdib [38, 39] is a Java encapsulation of the Verbs API through JNI, which increases Java communication performance using directly RDMA mechanisms. The main drawbacks of Jdib are its low-level API (like Verbs) and the JNI call overhead incurred for each Jdib operation (i.e., each function of the Verbs interface has to be wrapped through JNI). jVerbs [40] is a networking API and library for the JVM that offers RDMA semantics and exports the Verbs interface to Java. jVerbs maps the RDMA hardware resources directly into the JVM, allowing Java applications to transfer data without OS involvement. Although jVerbs is able to achieve almost bare-metal performance, its low-level API demands a high programming effort (as with Jdib). Additionally, jVerbs requires specific user drivers for each supported RDMA adapter, as the access to hardware resources in the data path is device-specific. Currently, it only supports some models and vendors (e.g., Mellanox ConnectX-2).

DOI: 10.1002/cpe

Regarding MPJ libraries, there have been several efforts to develop a message-passing framework since the inception of Java. Although the current MPI standard declaration is limited to C and Fortran languages, there have been a number of standardization efforts made towards introducing an MPI-like Java binding. The most widely used API has been proposed by the mpiJava [41] developers, known as the mpiJava 1.2 API [42]. Currently, the most relevant implementations of this API are Open MPI Java, MPJ Express, and FastMPJ, next presented.

mpiJava [41] consists of a collection of wrapper classes that use JNI to interact with an underlying native MPI library. However, mpiJava can incur a noticeable JNI overhead [43] and presents some inherent portability and interoperability issues derived from the amount of native code that is involved in a wrapper-based implementation (note that all the methods of the MPJ API have to be wrapped). More recently, Open MPI [44] has revamped this project and included Java support in the developer code trunk. The Open MPI Java interface is based on the original mpiJava code and integrated as a set of bindings on top of the Open MPI core [45].

MPJ Express [46] presents a modular design which includes a pluggable architecture of low-level communication devices that allows to combine the portability of the 'pure' Java shared memory device (`smpdev`) and New I/O (NIO) sockets communications (`niodev`), along with the native Myrinet support (`mxdev`) through JNI, implemented on top of the Myrinet eXpress (MX) library [47]. Additionally, the hybrid device (`hybdev`) allows to use simultaneously `niodev` and `smpdev` for inter-node and intra-node communications, respectively. Furthermore, the recently released native device [48] enables MPJ Express to exploit the latest features of native MPI libraries through JNI. However, the overall design of MPJ Express relies on an internal buffering layer that significantly limits performance and scalability [43].

Finally, FastMPJ [9] is our Java message-passing implementation that includes a layered design approach similar to MPJ Express, but avoiding its data buffering overhead by supporting direct communication of any serializable Java object. Moreover, FastMPJ includes a scalable collective library which implements up to six algorithms per collective primitive. More details about FastMPJ design and communications support are presented in Section 4.

This paper introduces new communication devices that provide efficient RDMA network support in the context of the Java language and the FastMPJ software. Previous MPJ middleware (e.g., mpiJava and MPJ Express) can also provide this specific support (i.e., not using TCP/IP emulations), but only when relying on an underlying native message-passing library. In fact, most of the contributions of the implemented Java communication devices have been motivated by the success of related works in native MPI libraries, where far more research has been performed. For instance, Liu *et al.* [49, 50] explored the feasibility of providing high-performance RDMA communications over InfiniBand in the context of the MPICH project [51]. Sur *et al.* [52] proposed several alternatives to exploit the RDMA Read operation in MVAPICH [53] for implementing an efficient long-message protocol over InfiniBand. The efficient support of custom Cray supercomputers (e.g., XT/XE/XK/XC) and their proprietary high-speed networks (e.g., SeaStar/Gemini/Aries) has also been an important research topic in the context of MPI libraries [54–56]. Our work tries to adapt all the research conducted in MPI to MPJ, taking into account the particulars of the Java language (e.g., buffer management and garbage collector).

4. OVERVIEW OF THE FASTMPJ COMMUNICATION DEVICE LAYER

Figure 2 presents a high-level overview of the FastMPJ design, whose point-to-point communication support relies on the `xxdev` device layer for interaction with the underlying hardware. This layer is designed as a simple and pluggable architecture of low-level communication devices that enables the incremental development of FastMPJ. Furthermore, it also eases the development of new `xxdev` devices reducing their implementation effort and minimizing the amount of native code needed to support a specific network through JNI, as only a very small number of methods must be implemented. Hence, it allows to combine the portability of 'pure' Java communication devices with high-performance network support wrapping native communication libraries through JNI. These `xxdev` devices abstract the particular operation of a communication protocol conforming to an API on top of which FastMPJ implements its communications. Therefore, FastMPJ communication

DOI: 10.1002/cpe

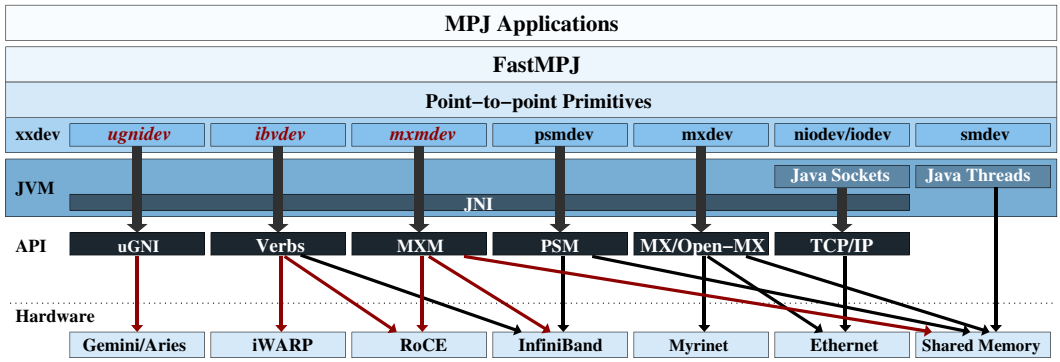


Figure 2. Overview of the FastMPJ communication devices.

devices must conform with the API provided by the abstract class `xxdev.Device` [9]. The low-level `xxdev` API only provides basic point-to-point communication methods and is not aware of higher level MPI abstractions like communicators. Thus, it is composed of basic message-passing operations such as point-to-point blocking and non-blocking communication methods, including also synchronous communications. The use of pluggable low-level devices for implementing the communication support is the most adopted approach in native message-passing libraries, such as the Byte Transfer Layer (BTL) and Matching Transport Layer (MTL), both included in Open MPI [44].

Among the main benefits of the `xxdev` device layer are its flexibility, portability, and modularity thanks to its encapsulated design. Furthermore, this layer supports the direct communication of any serializable Java object without data buffering. Hence, `xxdev` provides native devices (i.e., devices that implement the `xxdev` layer through JNI) with the buffer management of the Java arrays involved in a certain communication operation (either send or receive). In fact, this service can return a copy of the array using the *Get/Release[Type]ArrayElements()* family of JNI functions or a direct pointer to the contents of the array via *Get/ReleasePrimitiveArrayCritical()*. By using this service, specific implementations of native devices can potentially reduce some unnecessary data copies when possible (e.g., using blocking communications). Therefore, this fact allows `xxdev` communication devices to implement zero-copy protocols when communicating primitive data types using, for instance, RDMA-enabled networks.

Currently, FastMPJ includes three `xxdev` devices that support RDMA-enabled networks (see Figure 2): (1) `mxdev`, for Myrinet adapters and additionally for generic Ethernet hardware; (2) `psmdev`, for the InfiniPath family of IB adapters from Intel/QLogic; and (3) `ibvdev`, for IB adapters in general terms. These devices are implemented on top of MX/Open-MX, InfiniPath PSM, and Verbs native communication layers, respectively. Furthermore, the TCP/IP stack support is included through Java NIO (`niodev`) and IO (`iodev`) sockets, whereas high-performance shared memory systems can benefit from the thread-based device (`smdev`). The release of `niodev` as an open-source device is forthcoming.

As mentioned before, this paper presents two new `xxdev` communication devices, `ugnidev` and `mxmdev`, implemented on top of the user-level GNI (uGNI) and MXM native communication layers, respectively. The `mxmdev` device also includes efficient intra-node shared memory communication provided by MXM. An enhanced version of the `ibvdev` device, which extends its current support to RoCE and iWARP networking hardware and introduces an optimized short-message communication protocol, is also included. These communication devices (highlighted in italics and red in Figure 2) have been integrated transparently into FastMPJ thanks to its modular structure. Therefore, the developed devices allow current MPJ applications to benefit transparently from a more efficient support of RDMA networks (depicted by red arrows at the hardware level).

DOI: 10.1002/cpe

5. SCALABLE COMMUNICATIONS ON CRAY SUPERCOMPUTERS: `UGNIDEV`

The Cray XE/XK/XC family is nowadays an important class of custom supercomputers for running highly computationally intensive applications, with several systems ranked in the TOP500 list [12]. A critical component in realizing this level of performance is the underlying network infrastructure. As mentioned in Section 2, the Cray XE/XK architectures include the Gemini interconnect, whereas the newer XC systems are equipped with the Aries interconnect, both providing RDMA capabilities. Cray provides two low-level interfaces for implementing communication libraries targeted for these interconnects: GNI and DMAPP. In particular, the GNI API is mainly designed for applications whose communication patterns are message-passing in nature, while the DMAPP interface is geared towards Partitioned Global Address Space (PGAS) languages. Therefore, GNI would be the preferred interface on top of which a message-passing communication device as `ugnidev` should be implemented.

5.1. GNI API overview

The GNI interface exposes a low-level API that is primarily intended for the following: (1) kernel-space communication through a Linux device driver and the kernel-level GNI (kGNI) implementation; and (2) direct user-space communication through the user-level GNI (uGNI) library, where the driver is used to establish communication domains and handle errors, but can be bypassed for data transfer. Hence, the `ugnidev` device has been layered over the uGNI API, which provides two hardware mechanisms for initiating RDMA transactions using either Fast Memory Access (FMA) or Block Transfer Engine (BTE).

On the one hand, the FMA hardware provides in-order RDMA as a low-overhead, kernel-bypass pathway for injecting messages into the network, achieving the lowest latencies and highest message rates for short messages. Several forms of FMA transactions are available:

- FMA Short Messaging (SMSG) and FMA Shared Message Queue (MSGQ) provide a reliable messaging protocol with send/receive semantics that can be used for short point-to-point messages. These facilities are implemented using a specialized RDMA PUT operation with remote notification.
- FMA Distributed Memory (FMA DM) is used to execute RDMA PUT, GET, and atomic memory operations, moving user data between local and remote memory.

On the other hand, the BTE hardware offloads the work of moving bulk data from the host processor to the network adapter, also providing RDMA PUT and GET operations. The BTE functionality is intended primarily for long asynchronous data transfers between nodes. More time is required to set up data for a transfer than for FMA, but once initiated, there is no further involvement by the CPU. However, the FMA hardware can give better results than BTE for medium-size RDMA operations (2-8 KB), whereas BTE transactions can achieve the best computation–communication overlap because the responsibility of the transaction is completely offloaded to the network adapter, providing an essential component for realizing independent progress of messages. To achieve maximum performance, it is important to properly combine FMA and BTE mechanisms in the `ugnidev` implementation.

The memory allocated by an application must be registered with the network adapter before it can be given to a peer as a destination buffer or used as a source buffer for most uGNI transactions. Thus, in order to directly access a memory region on a remote node, the region must have been previously registered at that node. uGNI provides memory registration interfaces for the applications that allow to specify access permissions and memory ordering requirements. uGNI returns an opaque Memory Handle structure upon successful invocation of one of the memory registration functions. The Memory Handle (MH) can then be used for FMA/BTE RDMA transactions and SMSG/MSGQ messaging protocols. The registration and unregistration operations can be very expensive, which is an important performance factor that must be taken into account in the implementation of the `ugnidev` communication protocols.

DOI: 10.1002/cpe

Finally, uGNI also provides Completion Queues (CQ) management, as a lightweight event notification mechanism for applications. For example, an application may use the CQ to track the progress of local FMA/BTE transactions or to notify a remote node that data have been delivered to its memory. An application can check for presence of CQ Events (CQE) on a CQ in either polling or blocking mode. A CQE includes application-specific data, information about what type of transaction is associated with the CQE, and whether the transaction associated with the CQE was successfully completed or not. More specific details of the uGNI API can be found in [57].

5.2. FastMPJ support for Cray ALPS

Current Cray systems utilize the Cray Linux Environment (CLE), which is a suite of HPC tools that includes a Linux-based OS designed to run large applications and scale efficiently to a high number of cores. Hence, compute nodes run a lightweight Linux called Compute Node Linux (CNL) which ensures that OS services do not interfere with application scalability. Two separate execution environments for running jobs on the compute nodes of a Cray machine are currently available: Extreme Scalability Mode (ESM) and Cluster Compatibility Mode (CCM).

On the one hand, ESM is the high-performance and native execution environment specifically designed to run large applications at scale, which dedicates compute nodes for each user job and sets up the appropriate parallel environment automatically. This mode is required in order to access the underlying interconnect via the native uGNI API, thus allowing to obtain the highest network performance. However, ESM does not provide the full set of Linux services (e.g., ssh) needed to run standard cluster-based applications, which requires the implementation of specific support for this mode, as will be shown subsequently. On the other hand, the CCM execution environment allows standard applications to run without modifications. Thus, users can request the CNL on compute nodes to be configured with CCM through the use of a special queue at job submission. This mode comes with a standardized communication layer (e.g., TCP/IP) and emulates a Linux-based cluster which provides the services needed to run most cluster-based third-party applications on Cray machines. However, this feature is generally site dependent and may not be available. In addition, it poses important constraints such as that the number of cores that can be used under this mode is usually very limited, and there is no support for core specialization. Furthermore, the uGNI API cannot be used to directly access the underlying interconnect, which prevents the implementation of `ugnidev`. Therefore, a mandatory prerequisite for this device is the implementation of the ESM mode support in FastMPJ, which basically involves modifying the FastMPJ runtime to work in conjunction with the specific Cray scheduler, as described next.

The Application Level Placement Scheduler (ALPS) [58] is the Cray-supported mechanism for placing and launching applications under the ESM mode. More specifically, 'aprun' is the user command that must be used to launch a parallel application to a set of compute nodes reserved through ALPS. The FastMPJ support for Cray ALPS mainly consists of two distinct parts. The first one is the 'alps-spawner' utility, a small C program (< 400 source lines) intended to be launched with the 'aprun' command that acts as a bridge between ALPS and FastMPJ. This utility uses the C-based implementation of the Process Management Interface (PMI) [59], which is provided by Cray to interact with ALPS. The PMI library allows to obtain the necessary data from ALPS to properly set up the parallel environment of FastMPJ (e.g., rank of each process in the application). After setting this information via environment variables, 'alps-spawner' executes a new JVM using the *execvp()* function. Each JVM represents one of the Java processes of the MPJ application running a specific Java class of the FastMPJ runtime. This Java class, which is the second part of the implemented support, initializes the FastMPJ runtime with the information gathered from the environment and then invokes the main method of the MPJ application using the Java reflection facility. The MPJ application to be executed is one of the input parameters that are accepted by the 'alps-spawner' utility, which can be specified using both class and JAR file formats. Once the main method is running, the application will call at some point the `Init` method of the MPJ API in order to initialize the FastMPJ execution environment, and hence, the `ugnidev` device initialization takes place.

DOI: 10.1002/cpe

5.3. Initialization

Because the uGNI interface allows for user-space RDMA communication, there is a hardware protection mechanism to validate all RDMA requests generated by the applications. To utilize this mechanism, uGNI provides applications with a Communication Domain (CDM), which is essentially a software construct that must be attached to a network adapter in order to enable data transfers. Hence, processes must use a previously agreed upon protection tag (ptag) to define and join a CDM. For user-space applications, ALPS supplies a ptag value for each job together with the network adapter that the processes on the local node can use. This information is available in the `ugnidev` device initialization as part of the procedure described in the previous section, in which the required data is first obtained from ALPS/PMI and then is set up by the FastMPJ runtime.

Therefore, `ugnidev` first creates a CDM using the ptag value provided by the FastMPJ runtime and then attaches the CDM to the available network adapter. All the processes of the job must sign on to the CDM, as any attempt to communicate with a process outside of the CDM generates an error. In addition, each process must supply a 32-bit instance identifier which is unique within the CDM. The rank of the process within the global MPJ communicator (i.e., MPI.COMM_WORLD) is used for this purpose. After this step, `ugnidev` is able to create the CQs and register memory with the CDM. Having completed this sequence of steps, all processes can initiate communications. These operations are all asynchronous, with CQEs being generated when an operation or sequence of operations has been completed.

5.4. Communication protocols

The `ugnidev` device implements all its communication routines as non-blocking primitives through native methods in JNI. Therefore, blocking communication support is implemented as a non-blocking primitive followed by a *wait*-like call. Note that the current implementation of the `ugnidev` communication protocols does not make use of any additional thread (i.e., message progression for pending non-blocking communication requests occurs, if needed, when any `ugnidev` method is invoked). A message in `ugnidev` consists of a header plus user data (or payload). The message header includes the source identifier, the message size, the message tag, and control information (e.g., message type).

As mentioned in Section 5.1, two mechanisms are provided to transfer data using uGNI: FMA and BTE. It is clear that efficiently transferring message data requires to select the best mechanism based on the message size and the overhead associated with each one. Thus, the `ugnidev` device implements two different communication protocols, which are widely used in message-passing libraries:

1. Eager protocol: the sending process eagerly sends the entire message to the receiver, on the assumption that the receiver has available storage space. This protocol is used to implement low-latency message-passing communications for short messages (Section 5.5).
2. Rendezvous protocol: this protocol negotiates, via special control messages, the buffer availability at the receiving side before the message is actually transferred. This protocol is used for transferring long messages, whenever the sender is not sure whether the receiver actually has enough buffer space to hold the entire message (Section 5.6).

The maximum message size that can be sent using the eager protocol is a configurable runtime option of `ugnidev` that serves as a threshold for switching from one protocol to another. By default, the value of this threshold is set to 16 KB. The benefits of these protocols on the performance of MPJ applications can be significant. On the one hand, the eager protocol reduces the start-up latency, allowing Java applications with intensive short-message communications to increase their scalability. On the other hand, the rendezvous protocol maximizes communication bandwidth, thus reducing the overhead of message buffering and network contention.

DOI: 10.1002/cpe

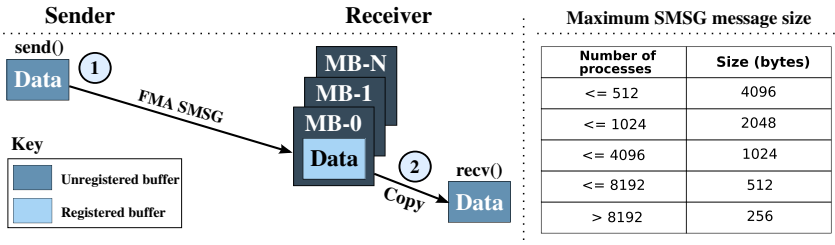


Figure 3. First path of the eager protocol in `ugnidev`.

5.5. *Eager protocol*

The eager protocol of `ugnidev` has been implemented using two different paths depending on the message size. The first path uses the FMA SMSG facility, as it provides the highest performance in terms of latency and short-message rates, but comes at the expense of memory usage. Although the FMA MSGQ messaging protocol can be more scalable in terms of memory usage, it was discarded because it provides lower performance than SMSG, particularly in terms of short-message rate. Additionally, the maximum message size that can be sent using MSGQ is limited to 128 bytes. In theory, SMSG can be used to deliver messages up to 64 KB, but owing to memory footprint constraints and performance considerations, the practical upper limit is usually lower.

Figure 3 shows the operation of the first path using the FMA SMSG facility. In this path, each process creates and registers with the network adapter per-process destination buffers called mailboxes (MB in the figure). During a message transfer, the sender directly writes data to its designated mailbox at the receiving side (step 1 in Figure 3). Next, the received data is copied out from the mailbox to the application buffer provided by the user (step 2). SMSG handles the delivery to the remote mailbox and raises both a local and a remote CQE on the sending and receiving sides, respectively, upon successful delivery. Note that SMSG transactions are implemented internally by the Gemini/Aries hardware as a special class of RDMA PUT operations which require remote buffer memory registration (i.e., the mailbox), but not local memory registration, which allows to send the data directly from the unregistered application buffer to the destination mailbox, as depicted in Figure 3 (see the color key). Furthermore, SMSG allows to specify a header separately from the message payload to be sent. Every send request of `ugnidev` has been defined with a small buffer (16 bytes) that contains the message header, which is not shown in the figure for clarity purposes. However, using the SMSG protocol requires a significant amount of registered memory resources, which scale linearly with the number of processes in the job. To alleviate this problem, SMSG is only used for communications up to a certain short message size, which is a configurable runtime option. By default, the maximum message size that can be sent using SMSG varies with the job size, with smaller mailboxes being used as the job size increases, in order to decrease the amount of memory used for SMSG mailboxes for larger jobs (see table in Figure 3).

Above the maximum message size used by the FMA SMSG path, but below the rendezvous limit (16 KB by default), `ugnidev` switches to the second path that is implemented using both FMA DM and BTE mechanisms. These mechanisms require the memory addresses and handles of the send and receive buffers. Therefore, the second path uses a small shared pool of pre-registered buffers as opposed to the per-process mailboxes of the FMA SMSG path. Each buffer in the pool is large enough to contain an entire eager message. The buffers are used in a copy in/out fashion (from/to application buffer), as the overhead of data copies is relatively small for short messages. Because the entire pool is pre-registered during the initialization of the `ugnidev` device, there is no additional registration overhead for each message. Figure 4 illustrates the operation of the second path. As can be seen in the left part of the figure, the sending process reserves one buffer from the pool and copies the user data in it (step 1). Next, a control message (EAGER_GET_INIT) that includes buffer information is sent to the receiver through the FMA SMSG path (step 2). All control messages of `ugnidev` are short enough to be sent using the SMSG path. Once the receiving side has processed the control message, a buffer is reserved from the pool and, based on the message size, either FMA

DOI: 10.1002/cpe

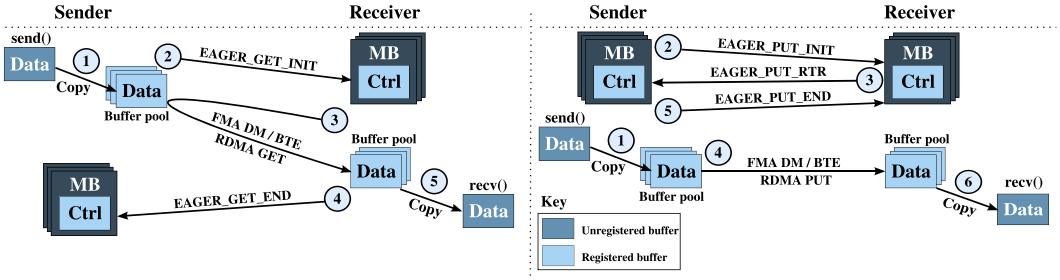


Figure 4. Second path of the eager protocol in `ugnidev`.

DM or BTE is used to initiate an RDMA GET of the message data from the sender's memory (step 3). Once the receiving process completes the GET operation, it sends an EAGER_GET_END message to the sender to complete the message transfer (step 4). Upon receipt of this message, the sender marks the message as complete and puts the buffer back to the pool. The receiver will copy the data out from the buffer in the pool to the application buffer when a *recv* operation matches the corresponding *send* (step 5). The choice between using FMA DM or BTE is also configurable via a runtime option. By default, messages up to 2 KB are sent using the FMA DM hardware, while BTE is more suitable for longer transfers, as mentioned in Section 5.1.

However, current Gemini/Aries network adapters impose some buffer size and alignment restrictions when using GET operations. More specifically, transfers using RDMA GET require that the size of the data buffer at both sides be a multiple of 4 and its start address be 4-byte aligned. When these restrictions are not met, `ugnidev` uses a PUT-based eager protocol (see right part of Figure 4). Hence, if the violation of these restrictions occurs at the sending side, an EAGER_PUT_INIT message is used after step 1 to express the intent to send an eager message using the PUT-based protocol (step 2). When the receiver has processed this message and has taken a buffer from the pool, it replies to the sender with an EAGER_PUT_RTR message to express that is ready to receive the data (step 3). Upon receiving this message, the sender uses the buffer information included in the control message to send the data using an RDMA PUT operation (step 4). If the restrictions are met at the sending side (i.e., the sender is using the GET-based protocol shown in the left part of the figure), but the violation occurs at the receiving side (e.g., the data size to be received is not a multiple of 4), the receiver sends an EAGER_PUT_RTR message to the sender in response to the EAGER_GET_INIT, including information of the receive buffer. This control message causes the sender to switch to the PUT-based protocol, using RDMA PUT to send the data (step 4). Once the PUT operation is complete, the sender sends an EAGER_PUT_END message in order to indicate the completion of the message transfer at the receiving side (step 5). Thus, the receiver is ready to copy the data out from the buffer in the pool to the application buffer if the corresponding *recv* operation has been issued (step 6).

One clear advantage of the GET-based protocol over the PUT-based is that the latter requires one extra control message, which increases the protocol overhead. Furthermore, the GET-based protocol can offer better computation–communication overlap, because the receiver can progress independently of the sender once the EAGER_GET_INIT message is sent. In order to achieve the lowest latency for short messages, the GET-based protocol is always used when possible, whereas the PUT-based path only acts as a fallback protocol due to the alignment restrictions of GET operations.

5.6. Rendezvous protocol

The rendezvous protocol is used for delivering messages exceeding the eager message size threshold. When transferring long messages, it is extremely beneficial to avoid extra data copies through a zero-copy protocol. The zero-copy protocol of `ugnidev` first negotiates the buffer availability at the receiving side using control messages. Thus, the application buffers are registered on-the-fly and the buffer addresses are exchanged via control messages. However, the actual data can be transferred by using either RDMA GET or PUT. The efficiency of the RDMA GET operation in Gemini/Aries is sensitive to the alignment of the send and receive buffers, and better performance is obtained

DOI: 10.1002/cpe

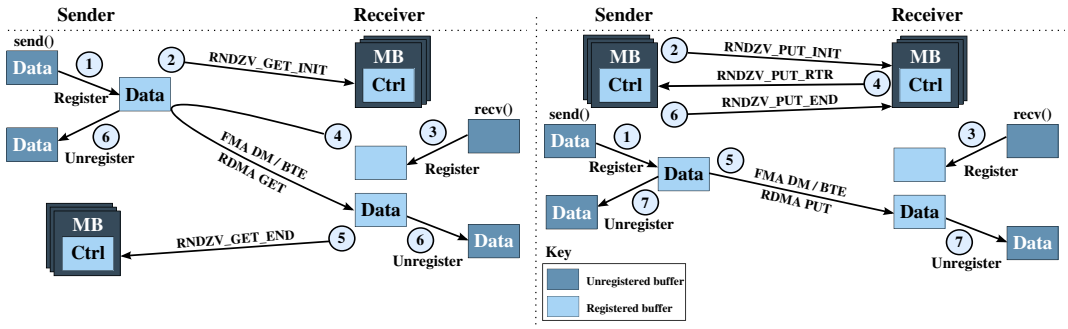


Figure 5. Rendezvous protocol implementation in `ugnidev`.

when these buffers start at the same relative offset into a cache line. However, RDMA PUT operations are much less sensitive to alignment and thus usually provide higher bandwidth than RDMA GET, especially for the long messages used in the rendezvous protocol. Hence, `ugnidev` employs a GET-based path up to a certain message size in order to benefit from its better computation–communication overlap capabilities and a PUT-based path for longer transfers. The threshold for switching from using RDMA GET to PUT is also a configurable runtime option, set to 64 KB by default. Additionally, the PUT-based path must also be used when buffer size and alignment restrictions of GET operations are not met, as occurred in the eager protocol.

In the GET-based path (left part of Figure 5), when a sending process is ready to send a long message, it first registers the application buffer (step 1) and then sends a RNDZV_GET_INIT message to the receiving process (step 2). This control message, in addition to expressing the intent to send the message, also provides the receiver with information of the send application buffer for performing an RDMA GET operation. Once the receiver is prepared to receive the message (i.e., the corresponding *recv* operation has been issued, and the receive buffer has been registered in step 3), an RDMA GET operation is initiated to access the message data directly from the send buffer (step 4). As in the eager protocol, the GET operation can be performed using either the FMA DM or the BTE hardware, depending on the value of the corresponding threshold. Using the default settings, all rendezvous transactions select BTE, as it generally provides better performance for long messages. Next, a RNDZV_GET_END message is sent to the sending process once the GET operation has finished at the receiving side (step 5). Finally, buffers are unregistered at both sides (step 6).

The PUT-based path (right part of Figure 5) is implemented as a seven-step protocol which starts when the sending process sends a RNDZV_PUT_INIT message to the receiver after the registration of the send buffer (steps 1 and 2). Once the receiver is prepared to receive the message, it registers the application buffer (step 3) and replies with a RNDZV_PUT_RTR message to express that is ready to receive the data (step 4). This reply message contains the information of the receive application buffer to access that memory region. Upon receiving this control message, the sender directly writes data to the target receive buffer by using a PUT operation (step 5). After this operation is finished, a RNDZV_PUT_END message is sent to indicate the completion of the message transfer at the receiving side (step 6), and finally, buffers can be unregistered (step 7).

5.7. Registration cache

When using the rendezvous protocol, application buffers are registered/unregistered on-the-fly, causing a performance penalty, especially for very long-message transfers. However, the overhead of the memory registration/unregistration can be hidden or at least reduced by using the pin-down cache technique [60]. The idea is to maintain a cache of registered buffers; thus, when a buffer is first registered, it is put into the cache, and when the buffer is unregistered, the actual unregister operation is not carried out and the buffer stays in the cache. Hence, the next time the buffer needs to be registered, no operation is performed because it is already in the cache.

The registration cache of the `ugnidev` device is implemented as a special Last In-First Out (LIFO) stack, which can also be linearly traversed. By adding new elements at the top of the stack,

DOI: 10.1002/cpe

buffers that are frequently used by a certain application can be found faster than those buffers that are rarely used. Hence, the effectiveness of this technique heavily depends on how often the application reuses its buffers and the way it does. If the reuse rate is high, most of the buffer registration and unregistration operations can be avoided. Moreover, if buffers are reused in an efficient way (i.e., first trying to reuse recently used buffers), the cost of the linear search is significantly reduced as they would be at the top positions. Nevertheless, the Gemini/Aries hardware imposes strong restrictions on the number of buffers that can be registered by user applications. Hence, the size of this cache has been limited, which also helps to decrease the cost of the linear search, especially for the worst-case scenario (i.e., the requested buffer is the last element). Furthermore, the cache is bypassed when it is already full and a certain number of consecutive cache misses has been reached in order to likely avoid costly and useless searches. By default, the `ugnidev` device uses the registration cache, but it can be disabled via a configurable runtime option.

6. EFFICIENT SUPPORT FOR RDMA ADAPTERS BASED ON VERBS: `IBVDEV`

The `ibvdev` device is a low-level message-passing device for communication on InfiniBand (IB) systems. This device directly implements its communication protocols on top of the Verbs interface through JNI. An initial proof-of-concept implementation of `ibvdev` was first integrated into the MPJ Express library [10] for internal testing purposes, but was never part of the official release. Although it was able to provide better performance than using the IPoIB protocol, the buffering layer in MPJ Express significantly limited its performance and scalability. Next, the `ibvdev` device was reimplemented to conform with the `xxdev` API and adapted for its integration into the FastMPJ middleware in order to improve its performance.

However, the `ibvdev` device still presents two important limitations: (1) it does not include support for the RDMA Communication Manager (RDMA CM), relying instead on TCP sockets to exchange the necessary information for establishing the initial connections between processes during the initialization method. This causes `ibvdev` to only work on IB adapters, thus not supporting the remaining RDMA-compliant adapters based on the Verbs interface: iWARP and RoCE. And (2) it does not take advantage of the inline feature that is provided by some RDMA adapters to improve the latency of short messages. Currently, `ibvdev` has overcome these constraints by establishing initial connections through RDMA CM and implementing a more efficient eager protocol that uses the inline feature. The new connection setup using RDMA CM allows `ibvdev` to support iWARP and RoCE networks while avoiding any TCP processing overhead during the initialization method. These new features in `ibvdev` will be discussed in the next sections.

6.1. Eager protocol optimization

The `ibvdev` device implements both the eager and rendezvous protocols relying on the Reliable Connection (RC) transport service defined in Verbs, which provides reliability, delivery order and data loss and error detection. The eager protocol of `ibvdev` is illustrated in Figure 6. In the original implementation, the buffer registration/unregistration overhead is avoided by using a shared pool of pre-registered, fixed size buffers for communication. For sending an eager message, the user data along with the message header are first copied to one of the available buffers from the pool (step 1 of the figure). Next, it is sent out from this buffer to the Send Queue (SQ) of the corresponding Queue Pair (QP). This is performed by using the *ibv_post_send()* function (step 2), which posts a Work Request (WR) to the SQ. At the receiving side, a number of buffers from the pool are pre-posted in the Receive Queue (RQ) using *ibv_post_recv()* (step 0). This function, which posts a WR to the RQ, is the receiving counterpart of *ibv_post_send()*. Once the message is received through the network (step 3), the message payload is copied out to the user destination buffer (step 4), and the receive buffer is returned back to the pool.

However, this implementation does not take advantage of sending data as inline, a feature that is supported by some modern RDMA adapters. Using this feature, the memory buffer that holds the message is placed inline in the WR posted to the SQ. This means that the CPU (i.e., not the RDMA adapter) will read the data from the buffer. Therefore, the data is transferred to the adapter at the

DOI: 10.1002/cpe

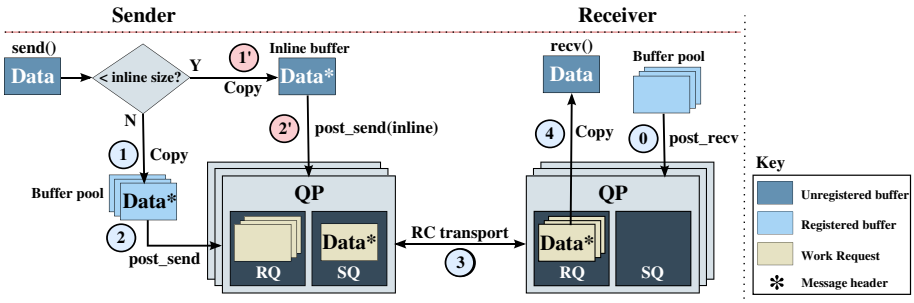


Figure 6. Eager protocol implementation in `ibvdev`.

same time that *ibv_post_send()* transfers the WR. The main benefit is that sending short messages as inline provides lower latency, because it eliminates the need of the RDMA device to perform an extra read over the PCIe bus in order to read the message payload. In addition, the memory buffer used for communication at the sending side does not have to be registered with the RDMA adapter.

The inline feature is an implementation extension not defined in the RDMA specification. Hence, there is not any defined *verb* that specifies the maximum message size that can be sent inline in the SQ of a QP. In some RDMA adapters with this feature, creating a QP will set the value of the *max_inline_data* attribute to the message size that can be sent as inline (usually less than 1 KB). In other adapters, the message size to be sent inline must be explicitly specified before the creation of a QP. In the latter case, the maximum value supported by the RDMA adapter is calculated during the initialization method of `ibvdev` following an iterative approach, which first creates a dummy QP specifying a high initial value and then continues to decrease if the QP creation fails. When the QP creation is successful, the inline size of the dummy QP is used to create all the QPs needed for establishing the connections between processes.

In the original implementation of `ibvdev`, when a WR is posted to the SQ, the buffer that holds the message cannot be modified, because it is not possible to know when the RDMA adapter will stop reading from it. That is to say, the WR is considered outstanding until a completion event is raised, which means that the buffer can now be reused. However, when using inline data, the buffer can be reused immediately after *ibv_post_send()* is finished, because the data has been already transferred to the RDMA adapter. This allows to have a single dedicated buffer to send inline data to all processes. Therefore, the pool of pre-registered buffers can be bypassed when using the new implemented path: if the message is short enough to be sent inline, the message header and payload are now copied to a dedicated buffer (step 1' in Figure 6) and then sent out from this buffer to the SQ using *ibv_post_send()* with the appropriate flags (step 2'). As mentioned before, this path reduces the latency of short messages, between 15–30% according to our tests, depending on the underlying RDMA adapter and CPU being used (Section 8.1.2). Additionally, it allows more buffers to be available to send messages through the original path if the message size is above the inline value, but below the rendezvous limit. Furthermore, all control messages of the rendezvous protocol can take advantage of this optimization, as they are small enough to be sent inline. This also contributes to increase the number of outstanding WRs that can be posted to the SQ at a time, which improves the overall efficiency of the RDMA adapter, while memory consumption remains almost the same (only one additional buffer is needed). Note that this optimized path is only relevant at the sending side, as the receiver is not aware of the fact that a WR is sent inline.

6.2. *RDMA CM-based connection setup*

The basic communication in `ibvdev` is achieved over connected QPs using the RC transport service. In the initialization method, an RC-based QP connection is established between every two processes (see Figure 6). To enable data transfers, each QP needs to be set up and must be transitioned through an incremental sequence of states. In order to transition into the final connected state, some information from the remote process is required: (1) the number of the remote QP to connect with (this value is returned at QP creation); and (2) the Local IDentifier (LID) of the remote process,

DOI: 10.1002/cpe

which is a unique 16-bit address assigned to end nodes by the subnet manager. This information needs to be exchanged through some out-of-band mechanism. As a first step, the original initialization method of `ibvdev` uses sockets to set up a TCP connection between every two processes. Second, the necessary information is exchanged through TCP sockets. Third, the QPs are transitioned and connected to each other. Finally, the TCP connection is closed. The described connection setup works perfectly on IB adapters, which was the main goal of the original implementation of the device. However, it poses an important drawback: the iWARP protocol requires RDMA CM to establish connections, which prevents `ibvdev` from working on iWARP adapters. Another drawback is the additional TCP connection that is established in advance to initialize the device, which can add a noticeable delay and TCP processing overhead on IB adapters when using a high number of processes. These issues have been overcome by implementing an alternative connection method using RDMA CM.

RDMA CM is an abstraction layer for connection management defined by the OFA [20], designed to establish connections between the QPs of a pair of processes. In fact, it is an event-driven connection manager based on a high-level IP address/port number abstraction that can set up connections over the multiple RDMA networks supported by Verbs, but is only mandatory for iWARP. The main responsibilities of RDMA CM include exchanging necessary connection information and transitioning the QPs through their states into the connected state, thus avoiding the additional TCP connection of `ibvdev`. It is to be noted that RDMA CM sets up the connections in a traditional client-server mechanism. Therefore, the API is based on sockets, but adapted for QP-based semantics: communication is over a specific RDMA device, and data transfers are message-based. RDMA CM provides only the communication management functionality (i.e., connection setup and teardown) and works in conjunction with the Verbs interface for data transfers.

The initialization method of `ibvdev` has been modified to use the RDMA CM manager in order to automate and simplify the connection setup. As mentioned before, RDMA CM uses the traditional TCP style, client-server mechanism to set up connections. Due to this, all the process pairs need to be separated into client–server pairs before any setting up of connections. For every pair of processes, the process with the lower rank takes the role of server (passive side or responder), and the process with the higher rank takes the role of client (active side or initiator). The main steps to complete the connection setup using RDMA CM are shown in Figure 7, as follows. (1) Each process identifies the port and IP address based on the RDMA adapter to use. This is accomplished via the information provided by the FastMPJ runtime to the `ibvdev` device. (2) Both sides allocate a communication identifier via *rdma_create_id()*, which is conceptually equivalent to a socket for RDMA communication. (3) The server must bind the RDMA identifier to a source address and listen for incoming connection requests. In the case of a client, it first resolves the server address and then allocates a new RDMA connection (i.e., a QP) via the *rdma_resolve_addr()* and *rdma_create_qp()*

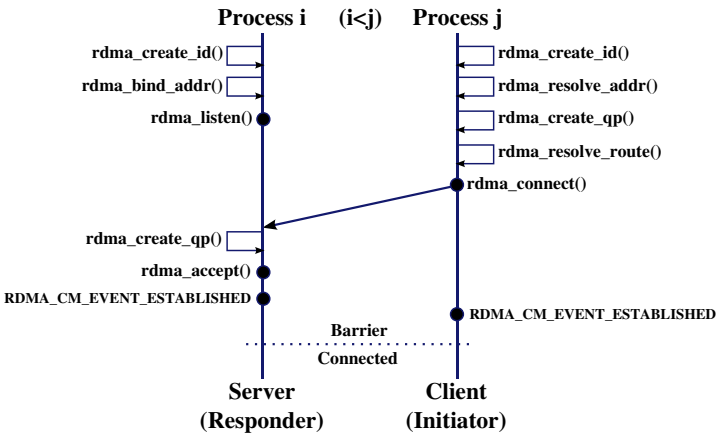


Figure 7. RDMA CM-based connection setup in `ibvdev`.

DOI: 10.1002/cpe

functions, respectively. (4) The client sends a connection request to the server using *rdma_connect()* after having resolved the destination route. (5) When the request is received at the server side, the responder then allocates a new RDMA connection and uses *rdma_accept()* to confirm the connection to the client. (6) The connections are established internally by RDMA CM, exchanging the necessary information and transitioning the corresponding QPs through their sequence of states. (7) The final transition into the connected state is detected via an event at both sides, which completes the establishment of the RDMA connection. (8) At this point, the processes synchronize with a barrier to make sure that all the peer processes are ready for communication. These steps are repeated for the setup of each of the QPs between a pair of processes. The overall procedure can be performed concurrently due to the event-driven nature of the connection manager.

As mentioned before, the RDMA CM-based connection setup allows `ibvdev` to provide support for iWARP adapters while leveraging the existing communication protocols of the device. Additionally, as RDMA CM is also valid for RoCE adapters, `ibvdev` now supports all RDMA-compliant adapters based on Verbs: IB, iWARP, and RoCE. The original TCP-based connection setup is still interesting to be supported, as it serves as a fallback option in case of any issue with RDMA CM, or even if it is not available in the system. Although the TCP-based approach cannot work on iWARP because RDMA CM is mandatory for this network, its support for RoCE has also been implemented, as described next.

The OFA specifies that Verbs applications which run over IB/iWARP should work on RoCE as long as the Global Routing Header (GRH) information is provisioned when creating Address Handles (AH). The GRH is required for routing between subnets and is optional within IB/iWARP subnets. However, RoCE encapsulates the IB transport and GRH headers in Ethernet packets bearing a dedicated *ether* type. In this case, the GRH is used for routing inside the subnet and therefore is mandatory. The GRH information can be provisioned in the AH of a QP when using the RC transport. The AH describes the path to the remote QP and is needed to make the transition from the initial state to the ready-to-receive state. This is the reason why using RDMA CM works seamlessly on RoCE without any change (QPs are transitioned and set up automatically). However, using the original TCP-based method, the GRH information must be specified manually using the Global IDentifier (GID) of the remote process, which is a unique 128-bit address used to identify a port on a network adapter that is assigned by the subnet manager. Hence, this method has been modified as follows. First, each process has to query its GID via *ibv_query_gid()*. Next, this value needs to be exchanged with the remote process along with the previously required information (LID and QP number). Once the TCP communication phase has been completed, the required GRH information for RoCE can be provisioned in the AH of each QP using the remote GID value. Finally, each QP can be transitioned through the required sequence of states as occurred in the original TCP-based implementation.

To sum up, the `ibvdev` device currently provides full support for IB, iWARP, and RoCE through the new RDMA CM-based connection setup and, as a fallback option, IB and RoCE are also supported using the TCP-based approach.

7. A SPECIFIC DEVICE FOR MELLANOX RDMA ADAPTERS: `MXMDEV`

Another contribution of this paper is the introduction of the `mxmdev` device, which provides native support for the networking infrastructure provided by Mellanox RDMA hardware over the MXM accelerator. MXM is a user-space messaging library that implements intra-node shared memory and inter-node communication protocols, which are completely transparent to the application. It includes a variety of enhancements that take advantage of Mellanox IB/RoCE adapters including proper management of resources and memory structures, efficient memory registration, handling of transport services, and a tag matching mechanism at the receiving side. Hence, many of the low-level network features are built-in in MXM, which allows developers to work at a higher level and the main effort to be spent on the overall application development.

Therefore, the most important benefit of MXM is that it provides the developer with a higher-level API than Verbs, based on a set of communication primitives with messaging semantics that eases the development of applications on top of the Mellanox RDMA hardware. However, MXM is not

DOI: 10.1002/cpe

primarily intended for use by end-user applications. Instead, portable communication middleware (e.g., message-passing libraries) usually provide specific support for MXM, which allows the user to benefit from a higher level of abstraction without source code modifications. This fact has motivated the implementation of `mxmdev`, a new `xxdev` device layered on top of the MXM library. FastMPJ with `mxmdev` provides the programmer with all the high-level features of the MPJ layer (e.g., collective communications, virtual topologies, intra-communicators and inter-communicators) while taking advantage of the infrastructure provided by FastMPJ, such as the runtime system. Furthermore, it frees Java developers from the implementation of JNI calls, which is usually a cumbersome and time-consuming development task. Hence, the `mxmdev` device allows the developer to benefit from the MPJ programmability, which greatly enhances productivity without trading off much performance.

7.1. Connection setup

The MXM library is initialized using the *mxm_init()* method. Next, the connection setup must be carried out in order to enable communications. In MXM, messages are exchanged between endpoints, which are software representations of the Mellanox IB/RoCE adapters. At present, MXM does not include any communication manager to ease the connection setup. Thereby, in order to establish the initial connections between endpoints, the `mxmdev` device has to rely on an out-of-band mechanism to distribute the endpoint addresses between all the processes. Hence, each process first creates and sets up an endpoint using the *mxm_ep_create()* function. After initializing endpoints, a Matched Queue (MQ) interface is created via *mxm_mq_create()*. Basically, an MQ is a specific context of sending and receiving messages which maintains ordering between requests. It exposes a simplified messaging interface that resembles an MPI communicator, but supporting only basic point-to-point communications. Next, the endpoint addresses are exchanged between all processes relying on TCP sockets, selected as the ubiquitous out-of-band mechanism. Finally, the *mxm_ep_connect()* function must be used to establish the endpoint connections with the information gathered from the TCP communication phase, thus enabling data transfers.

7.2. Basic communication operation

The MXM library provides a C-based API which includes a small set of point-to-point communication primitives similar to those needed to implement the `xxdev` interface (see Section 4). Thus, `mxmdev` acts as a thin wrapper over the MXM library, so that the implementation of a method in `xxdev` generally delegates directly in a native method that performs the requested operation in MXM through JNI. Therefore, `mxmdev` deals with the marshaling and communication of Java objects, the JNI transfers, and the handling of MXM parameters by implementing a series of three steps: (1) get the associated parameters of the Java objects that are required for calling the corresponding function in MXM; (2) call the MXM function; and (3) save the results in the appropriate attributes of the Java objects involved in the communication. As a general rule, the caching of object references has been extensively used in the implementation of the JNI layer, thus minimizing the overhead associated with the JNI calls.

Every message operation in MXM, either sending or receiving, starts with a non-blocking communication request (e.g., *mxm_req_send()*). This request is queued by MXM, returning the control to `mxmdev`. Next, the `mxmdev` device is responsible for checking the successful completion of the communication operation using one of the supported mechanisms in MXM (e.g., *mxm_wait(), mxm_req_test()*). The MXM tag matching mechanism at the receiving side is based on a 32-bit value (*mxm_tag_t)*, which must be specified by both communication peers in order to deliver incoming messages to the right receive requests. The tag value specified by the programmer at the corresponding MPJ-level method (e.g., *MPI.COMM_WORLD.Send()*) is used for this purpose. Hence, incoming MXM messages are stored according to their MPJ tags to pre-posted receive buffers. In this case, note that, unlike the `ugnidev` and `ibvdev` devices, the underlying communication protocols are implemented internally by MXM. Currently, MXM includes both intra-node (via shared memory) and inter-node communication protocols, allowing MPJ applications to take full advantage of hybrid shared/distributed memory architectures, which are currently the most generally adopted solutions in HPC.

DOI: 10.1002/cpe

8. PERFORMANCE EVALUATION

This section presents a performance evaluation of the FastMPJ communication devices presented in this paper: `ugnidev`, `ibvdev`, and `mxmdev`. The experimental results have been obtained at the MPJ/MPI level in order to analyze the impact of their use on the overall middleware performance. Hence, FastMPJ (labeled as FMPJ in the graphs) has been evaluated comparatively with representative native and Java messaging middleware: Open MPI [44], Open MPI Java [45], and MPJ Express [46]. First, this section includes a micro-benchmarking of point-to-point communication primitives on several RDMA networks (Section 8.1). Next, the impact of the communication devices on the overall application performance of representative parallel codes is analyzed (Section 8.2).

8.1. Micro-benchmarking of MPJ/MPI point-to-point primitives

The goal of this micro-benchmarking is the comparative performance evaluation of MPJ/MPI point-to-point communications between two nodes across different RDMA networks (i.e., inter-node latency and bandwidth). This evaluation has been carried out using a representative micro-benchmarking suite, the Intel MPI Benchmarks [61], and our own MPJ counterpart, which adheres to its measurement methodology. Here, the metric shown is the half of the round-trip time of a ping-pong test for short messages (up to 1 KB), and the corresponding bandwidth for longer messages (up to 16 MB). In order to obtain optimized JIT compiled bytecode results, 20,000 warm-up iterations have been executed before the actual measurements. The results shown are the average of 10,000 iterations, although the observed standard deviations were not significant. The transferred data are byte arrays, avoiding the Java serialization overhead that would distort the analysis of the results, in order to present a fair comparison with MPI.

8.1.1. Experimental configuration. Two different systems have been used in the evaluation of point-to-point primitives. The first testbed consists of two nodes, each of them with one Intel Xeon E5-2643 quad-core Sandy Bridge-EP processor at 3.3 GHz and 32 GB of memory. These nodes have been used to evaluate three different RDMA networks: IB (Mellanox MT27500 4x FDR, 56 Gbps), RoCE (Mellanox MT27500, 40 Gbps), and iWARP (Intel NetEffect NE020, 10 Gbps). Hence, this testbed allows the evaluation of the `ibvdev` device on IB, RoCE, and iWARP, while `mxmdev` can be assessed on IB and RoCE. Regarding software configuration, the OS is Linux CentOS 6.4 with kernel 2.6.32-358 and the JVM is OpenJDK 1.7.0_25. Finally, the native communication layers are OFED driver 3.5-2 and MXM version 1.5.

The second testbed is the Hermit supercomputer installed at the High Performance Computing Center Sttutgart (HLRS), ranked #44 in the June 2014 TOP500 list [62]. This system is a petaflop Cray XE6 supercomputer with 113,664 cores and 126 TB of memory. More specifically, Hermit consists of 3552 compute nodes, each of them with 2 AMD Opteron 6276 16-core Interlagos processors at 2.3 GHz and 32/64 GB of memory. The nodes are connected via the custom Gemini interconnect [18], which allows the evaluation of the `ugnidev` device. This network has a 3D torus topology built from Gemini Application-Specific Integrated Circuits (ASICs) that provide two network adapters and a 48-port router. Hence, each ASIC connects two nodes to the network. In the ping-pong test, two adjacent nodes (i.e., connected to the same ASIC) have been used in order to report the lowest latencies and highest bandwidths for inter-node communications (i.e., results are shown using the minimum hop network count). Regarding software settings, this system runs CLE version 4.1.UP01 with kernel 2.6.32.59, which is an OS based on SUSE Linux Enterprise Server. The JVM is Oracle JDK 1.7.0_45, and the uGNI library version is 4.0-1. This supercomputer is also one of the systems selected for the analysis of performance scalability of parallel Java applications (shown in Section 8.2).

Regarding the messaging middleware under comparison, Open MPI 1.7.4 has been evaluated as a representative open-source native MPI implementation. This middleware has been specifically configured with the `openib` BTL, which is implemented over Verbs on IB, RoCE, and iWARP networks. Open MPI results using the `mxm` MTL, implemented over MXM, are not shown for clarity purposes, as we have checked that the `openib` BTL generally obtains better performance than

DOI: 10.1002/cpe

the mxm MTL. Furthermore, Open MPI can benefit from its specific support for the Cray machine, which allows to use the ESM mode and the uGNI library via the ugni BTL. Open MPI Java[§] has been evaluated under the same BTL settings, whereas MPJ Express results are shown using the native communication device (version 0.43) on top of Open MPI 1.7.4, thus taking advantage of RDMA networks and avoiding inefficient TPC/IP emulations (e.g., IPoIB).

8.1.2. Analysis of the results. Figure 8 shows point-to-point latencies and bandwidths on IB, RoCE, iWARP, and Gemini networks. The latency graphs (at the left) serve to compare short-message performance (up to 1 KB), whereas the bandwidth graphs (at the right) show long-message performance (up to 16 MB). Latency graphs of IB, RoCE, and iWARP networks show performance results both for the previous version of ibvdev without inline support (labeled as 'w/o inline') and the new device with inline support (labeled as 'w/ inline'). As can be observed, the improved eager protocol of ibvdev provides latency reductions of up to 30% and 15% for IB/RoCE and iWARP, respectively. As expected, these performance improvements are only noticeable when transferring messages up to the maximum message size which is configured to be sent as inline data (i.e., 128 and 32 bytes for IB/RoCE and iWARP adapters, respectively). From now on, only the performance results of the new version of the ibvdev device are considered for comparison purposes.

The performance results on IB reveal that both FastMPJ devices (i.e., ibvdev and mxmdev) obtain the lowest start-up latencies for MPJ, around 1 μs, showing an overhead reduction of approximately 43% and 63% compared with MPJ Express (1.75 μs) and Open MPI Java (2.7 μs), respectively, whereas native MPI latencies are around 0.82 μs. Regarding bandwidth, the ibvdev device obtains the best MPJ performance with up to 47 Gbps, only slightly lower than MPI (48 Gbps), whereas the maximum bandwidth for mxmdev is around 43 Gbps. Here, FastMPJ clearly outperforms the other Java middleware for long messages, achieving up to 2.9 times higher bandwidth for 8 MB messages than MPJ Express, which suffers significantly from the high overhead of its buffering layer. Open MPI Java incurs a noticeable overhead from 256 KB on, showing poor long-message bandwidths.

The analysis of the performance results on RoCE shows a very similar pattern. On the one hand, FastMPJ devices obtain slightly higher latencies than on IB, around 1.10 μs, significantly outperforming MPJ Express (1.83 μs) and Open MPI Java (2.8 μs). MPI latencies also slightly increase on RoCE, around 0.90 μs. On the other hand, the maximum bandwidths obtained by ibvdev and mxmdev are 36.6 and 35.7 Gbps, respectively, up to 2.3 times better performance than MPJ Express. In fact, FastMPJ bandwidths are quite close to MPI (37.2 Gbps) and to the limit for this networking technology (40 Gbps). Both Open MPI Java and MPJ Express incur once again a high overhead and buffering penalty, which affects especially long-message performance. These results confirm that RoCE is able to provide IB-like latencies on the Ethernet infrastructure, while the maximum bandwidth of RoCE adapters is increasing and approaching IB.

The start-up latencies on iWARP are relatively high, at least compared with those obtained on IB and RoCE. This fact suggests that the TCP/IP processing overhead seems to be the main performance bottleneck for short-message performance, even though it is offloaded onto the iWARP hardware. In fact, all MPJ middleware achieve very similar latencies, around 8 μs, while native MPI results are around 6 μs. The observed bandwidths for FastMPJ and MPI are quite similar, up to 9.2 Gbps, which allows ibvdev to outperform MPJ Express (8 Gbps) and Open MPI Java (6.7 Gbps). In this scenario, the iWARP network, with a theoretical maximum bandwidth of 10 Gbps, turns out to be the main performance bottleneck for FastMPJ and MPI bandwidth results.

Finally, the performance results on the Cray Gemini interconnect show that the start-up latencies of the ugnidev device are around 1.45 μs, only slightly higher than MPI (1.38 μs) and significantly lower than MPJ Express (4.2 μs) and Open MPI Java (4.9 μs). This means that FastMPJ is able to provide a reduction of the communication overhead for short messages of up to 65% and 70% compared with MPJ Express and Open MPI Java, respectively. It can also be observed that the performance increase of ugnidev for long-message bandwidth is up to 275% with respect

[§]Open MPI Java version 1.9a1r29129 has been used, which is a snapshot from the code trunk that is fully compliant with the mpiJava 1.2 specification.

DOI: 10.1002/cpe

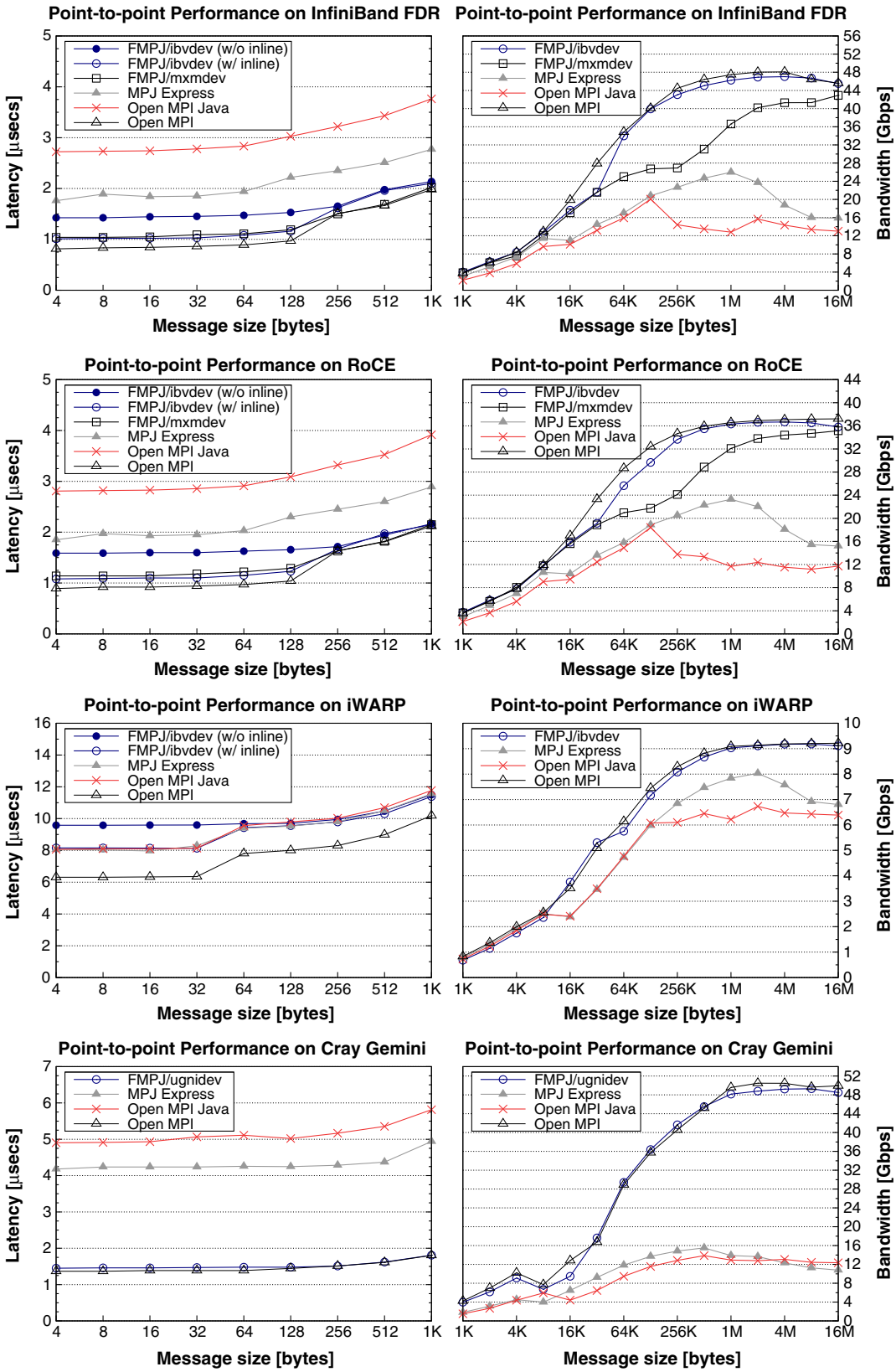


Figure 8. Performance of MPJ/MPI point-to-point communications on RDMA-enabled networks.

DOI: 10.1002/cpe

to the best MPJ alternative. This improvement is obtained for 4 MB messages, where FastMPJ achieves 49.2 Gbps, and Open MPI Java is limited to around 13.1 Gbps. Furthermore, this maximum bandwidth for FastMPJ is very close to that obtained by the native MPI library (50.4 Gbps).

This micro-benchmarking has shown significant improvements on the performance of MPJ point-to-point communications when using the Java devices presented in this paper. Moreover, FastMPJ results are even very close to those obtained by MPI. However, the usefulness of these devices depends on their impact on the overall application performance, as will be analyzed next.

8.2. Performance analysis of parallel codes

This section presents the performance analysis of representative HPC kernels and applications. On the one hand, the performance of two message-passing kernels selected from the NAS Parallel Benchmarks (NPB) implementation for MPI (NPB-MPI) [63] and MPJ (NPB-MPJ) [64] have been evaluated (Section 8.2.2): FT (Fourier Transform) and MG (Multi-Grid). On the other hand, the scalability of an MPJ version of the Finite-Difference Time-Domain (FDTD) method [65], which is a widely used numerical technique in computational electromagnetics, has been analyzed at the application level (Section 8.2.3). The selection of these parallel codes has been motivated by their high communication intensiveness, which allows the assessment of the impact of the developed communication devices on their scalability.

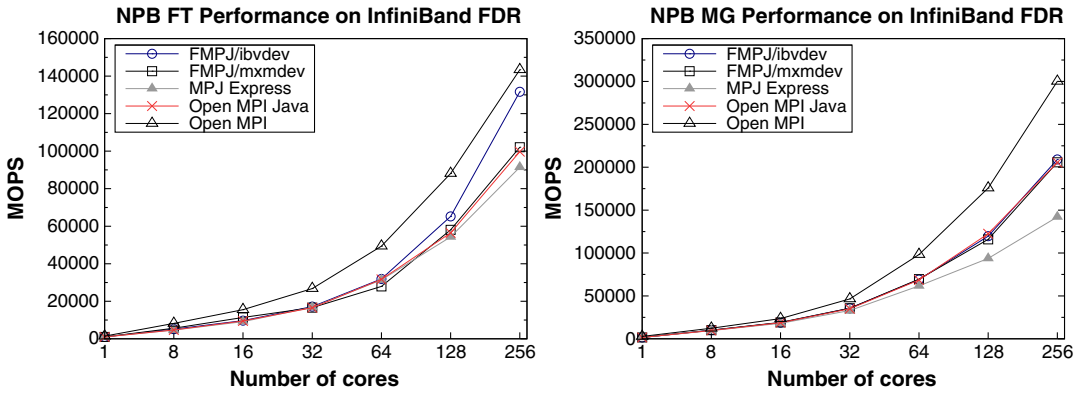
8.2.1. Experimental configuration. The experimental results have been conducted on two systems. The first testbed is Pluton, a 16-node multi-core cluster. Each node has two Intel Xeon E5-2660 octa-core Sandy Bridge-EP processors at 2.2 GHz (hence 16 cores per node) and 64 GB of memory. The performance results have been obtained using 16 processes per node (i.e., 256 cores in total), because we have checked that the use of 32 processes per node when resorting to the HyperThreading technology does not provide any performance benefit for the evaluated codes. The nodes of Pluton are interconnected via IB (Mellanox MT27500 4x FDR, 56 Gbps), which allows the assessment of the `ibvdev` and `mxmdev` devices. Regarding software configuration, the OS is Linux CentOS 6.4 with kernel 2.6.32-358 and the JVM is OpenJDK 1.7.0_25. Finally, the native communication layers are OFED driver 3.5-1 and MXM version 2.0.

The second testbed is Hermit, the Cray XE6 supercomputer described in Section 8.1.1. The AMD Opteron processor of Hermit has a quite complex architecture that provides up to 16 integer cores and eight 256-bit Floating Point Units (FPUs) per chip. A dual-processor node can provide up to 32 integer cores that access the half of the FPU executing 128-bit instructions or 16 integer cores accessing the entire FPU with 256-bit instructions. This is due to the sharing of each FPU between the two integer cores of a Bulldozer module, which is the building block of this architecture. Therefore, the results are shown using 16 processes per node (i.e., one process per Bulldozer module) in order to maximize the FPU performance on this system. We have experimentally checked that this configuration obtains the best performance for the evaluated codes, which carry out extensive double-precision floating-point operations, and thereby, the results using 32 processes per node are not shown for clarity purposes. Moreover, the reported results for a given application and core count were obtained within a single resource allocation to minimize timing differences due to node placement.

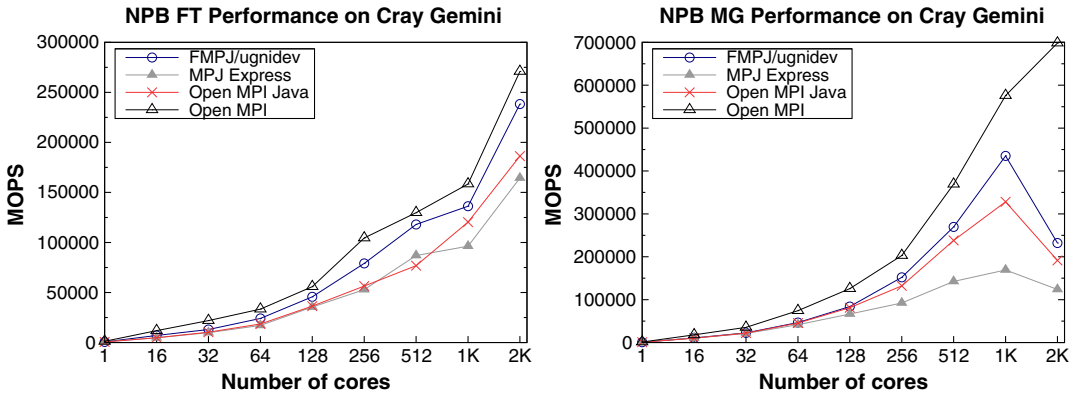
Regarding the messaging middleware, Open MPI has been configured on Pluton with the `openib` BTL for inter-node communications and the `sm` BTL for intra-node communications, whereas the `ugni` and `vader` BTLs have been used on Hermit for inter-node and intra-node communications, respectively. Hence, Open MPI Java and MPJ Express (using again the native communication device) have been evaluated with these BTL settings.

8.2.2. MPJ/MPI Kernel Performance Analysis. Figure 9 presents performance results for the FT and MG kernels on Pluton and Hermit using up to 256 and 2048 cores, respectively. In order to present a fair comparison between MPJ and MPI and provide a reference of the absolute NPB performance, the metric reported is Millions of Operations Per Second (MOPS), which refers to the number of operations performed in the kernel rather than the number of CPU operations. In fact,

DOI: 10.1002/cpe



(a) NPB results on the multi-core cluster (Pluton)



(b) NPB results on the Cray XE6 supercomputer (Hermit)

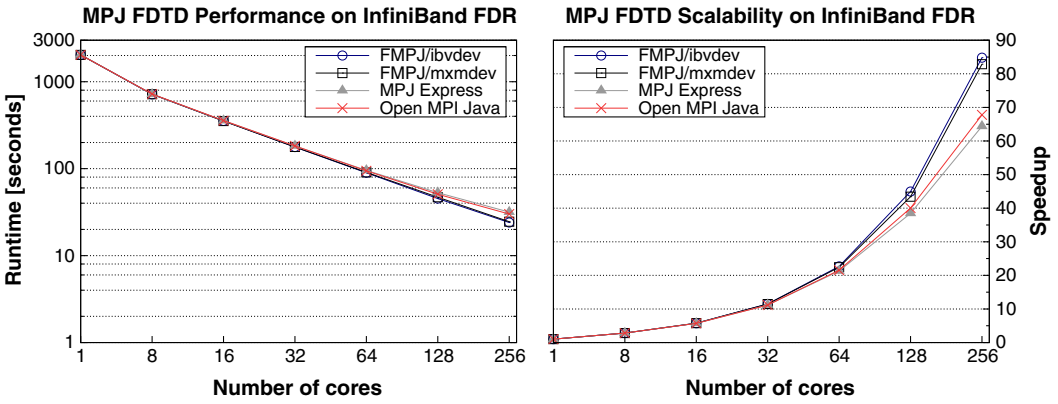
Figure 9. Performance of NPB-MPJ/MPI kernels.

NPB-MPJ/MPI results in terms of scalability should not be compared directly, as the sequential runtimes of the native and Java implementations for these kernels are different and thus, their parallel execution times. Hence, the longer runtime on one core for Java, whose performance is on average around 65% of the native counterpart, favors the scalability of the MPJ kernels (a heavy workload reduces the impact of communications on the overall performance scalability). In these experiments, the problem size is fixed using the NPB class C workload, while the number of cores is increased, hence applying a strong scaling model.

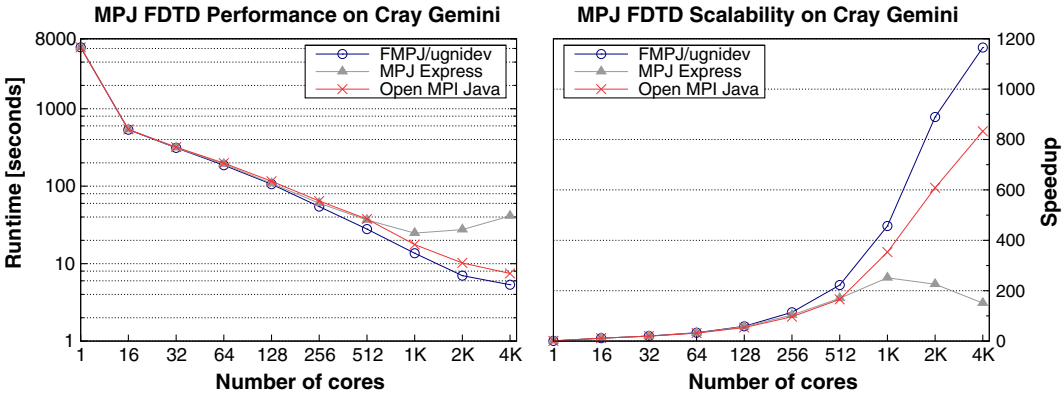
Regarding the results on Pluton (see Figure 9(a)), FastMPJ using `ibvdev` shows the best MPJ performance for the FT kernel from 32 cores on, outperforming Open MPI Java and MPJ Express up to approximately 31% and 43%, respectively. FastMPJ with the `mxmdev` device achieves a very similar performance to Open MPI Java. The native MPI library (Open MPI) obtains the highest MOPS for FT, a pattern that is maintained in the remaining experiments. However, FastMPJ results are highly competitive, obtaining up to 91% of the native MPI performance when using 256 cores. The reported MOPS for the MG kernel are quite similar for FastMPJ (using both devices) and Open MPI Java, which are around 70% of the MPI performance. This fact suggests that the memory access bandwidth turns out to be the main performance bottleneck for the Java code on this testbed. MPJ Express shows the poorest results, below 145,000 MOPS on 256 cores, which is around 32% lower than FastMPJ.

The analysis of the results on Hermit (see Figure 9(b)) shows that the use of the `ugnidev` device allows FastMPJ to become the best MPJ middleware for both kernels. Regarding FT results, FastMPJ clearly outperforms Open MPI Java and MPJ Express, especially from 64 cores on, providing a performance improvement of up to 28% and 45% on 2048 cores, respectively. FastMPJ

DOI: 10.1002/cpe



(a) MPJ FDTD results on the multi-core cluster (Pluton)



(b) MPJ FDTD results on the Cray XE6 supercomputer (Hermit)

Figure 10. Runtime and scalability of the FDTD parallel Java application.

results are again quite competitive when compared with MPI, around 87% of the native performance using the maximum number of cores. The peak values for the MG kernel are achieved on 1024 cores for all MPJ middleware, where `ugnidev` provides FastMPJ with a performance increase of 32% and 157% when compared with Open MPI Java and MPJ Express, respectively. In this case, FastMPJ obtains around 75% of the MPI performance when using 1024 cores. From this point, the performance of the MPJ codes degrades, whereas the MPI one continues to improve.

8.2.3. Performance Analysis of the MPJ FDTD application. Figure 10 shows the runtime and scalability results for the MPJ codes of the FDTD application on Pluton (Figure 10(a)) and Hermit (Figure 10(b)) using up to 256 and 4096 cores, respectively [¶]. This application simulates a Ricker wavelet propagating in free space surrounded by perfectly electrically conducting walls that reflect impinging electromagnetic waves. The parallel code is based on a domain decomposition approach that divides the workload equally among the cores, requiring frequent data transfers between processes during the entire simulation (mainly point-to-point communications). The results are shown for a simulation that consists of 2,500 time steps using a fixed 16384x8192 grid (i.e., strong scaling).

According to the reported results, FastMPJ achieves the highest speedups, as shown in the right graphs, especially when using a high number of cores. In particular, the performance improvements compared with Open MPI Java are 24% on Pluton and 40% on Hermit when using the highest core count on each testbed. Note that the `ibvdev` and `mxmdev` devices achieve very similar scalability results on Pluton. Regarding MPJ Express, the speedup increases provided by FastMPJ are 31%

[¶]MPI results are not shown due to the unavailability of the code.

DOI: 10.1002/cpe

on Pluton (for 256 cores) and 81% on Hermit (for 1024 cores). Note also that while MPJ Express obtains similar speedups to Open MPI Java on Pluton using up to 256 cores, it is not able to scale on Hermit when using more than 1024 cores.

The analysis of the results of this evaluation reinforce one of the main conclusions of this paper, that the use of efficient low-level communication devices can improve transparently the scalability of parallel Java applications.

9. CONCLUSIONS

RDMA is a well-known mechanism that enables zero-copy and kernel-bypass features, providing low-latency and high-bandwidth communications with low CPU utilization. However, RDMA-enabled networks also usually pose some important challenges (e.g., high programming effort) that require appropriate middleware support for the development of scalable parallel applications with underlying hardware transparency. In order to take full advantage of the abundant hardware resources due to the current trend of increasing the number of cores, applications have to resort to efficient middleware. Nevertheless, current Java communication middleware is usually based on protocols with high communication overhead which do not usually provide scalable communications on RDMA networks.

This paper has described in detail the implementation of several low-latency communication devices, which have been successfully integrated in our Java message-passing implementation, FastMPJ. These devices have considered several communication protocols in order to provide scalable support for RDMA networks, enabling 1-μs start-up latencies and up to 49 Gbps bandwidth for Java message-passing applications, thanks to the efficient exploitation of the underlying RDMA hardware. In order to evaluate the benefits of these devices, their performance has been analyzed comparatively with other Java and native communication middleware on representative RDMA networks (IB, RoCE, iWARP, and Cray Gemini) and parallel systems (a multi-core InfiniBand cluster and a TOP500 Cray supercomputer). The analysis of the results has demonstrated experimental evidence of significant performance improvements when using the developed devices in FastMPJ. In fact, the scalability of parallel Java codes can benefit transparently from this efficient support on RDMA networks, reducing the latency by up to 65% and 70%, and increasing the bandwidth by up to 3.9 and 3.7 times compared with MPJ Express and Open MPI Java, respectively. Furthermore, the analysis of the impact of the use of these devices on representative HPC kernels has shown that FastMPJ is able to obtain up to 87% of the native MPI performance on 2048 cores. Therefore, the reported advances in the efficiency of Java communications can contribute to increase the benefits of the adoption of Java for parallel computing, in order to improve productivity in parallel programming.

ACKNOWLEDGEMENTS

This work was supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds of the European Union (Project TIN2013-42148-P), by the Galician Government (Project GRC2013/055) and by the FPU Program of the Ministry of Education of Spain (FPU grant AP2010-4348). The authors thankfully acknowledge the High Performance Computing Center Stuttgart (HLRS) for providing access to the Cray XE6 supercomputer (Hermit) through a research visit to the Rechenzentrum Universität Mannheim (Germany) funded by an INDITEX-UDC grant. The authors would also like to thank Dr. Heinz Kredel and his group for making this research visit to Mannheim possible.

REFERENCES

1. Suganuma T, Ogasawara T, Takeuchi M, Yasue T, Kawahito M, Ishizaki K, Komatsu H, Nakatani T. Overview of the IBM Java just-in-time compiler. *IBM Systems Journal* 2000; **39**(1):175–193.
2. Shafi A, Carpenter B, Baker M, Hussain A. A comparative study of Java and C performance in two large-scale parallel applications. *Concurrency and Computation: Practice and Experience* 2009; **21**(15):1882–1906.
3. Taboada GL, Ramos S, Expósito RR, Touriño J, Doallo R. Java in the high performance computing arena: research, practice and experience. *Science of Computer Programming* 2013; **78**(5):425–444.
4. Apache Hadoop. Available from: http://hadoop.apache.org/ [accessed on 27 November 2014].

DOI: 10.1002/cpe

5. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Communications of the ACM* 2008; **51**(1):107–113.
6. Message Passing Interface Forum. MPI: a Message Passing Interface standard, 1995. Available from: http://www.mcs.anl.gov/research/projects/mpi/ [accessed on 27 November 2014].
7. Hongwei Z, Wan H, Jizhong H, Jin H, Lisheng Z. A performance study of Java communication stacks over InfiniBand and Gigabit Ethernet. *Proceedings of 4th IFIP International Conference on Network and Parallel Computing Workshops (NPC'07)*, Dalian, China, 2007; 602–607.
8. Carpenter B, Getov V, Judd G, Skjellum A, Fox G. MPJ: MPI-like message passing for Java. *Concurrency and Computation: Practice and Experience* 2000; **12**(11):1019–1038.
9. Expósito RR, Ramos S, Taboada GL, Touriño J, Doallo R. FastMPJ: a scalable and efficient Java message-passing library. *Cluster Computing* 2014; **17**(3):1031–1050.
10. Expósito RR, Taboada GL, Touriño J, Doallo R. Design of scalable Java message-passing communications over InfiniBand. *Journal of Supercomputing* 2012; **61**(1):141–165.
11. IBTA. InfiniBand trade association. Available from: http://www.infinibandta.org/ [accessed on 27 November 2014].
12. TOP500 Org. Top 500 supercomputer sites. Available from: http://www.top500.org/ [accessed on 27 November 2014].
13. RDMA consortium. Architectural specifications for RDMA over TCP/IP. Available from: http://www.rdmaconsortium.org/ [accessed on 27 November 2014].
14. IBTA. InfiniBand architecture specification release 1.2.1 annex A16: RDMA over Converged Ethernet (RoCE) [accessed on 27 November 2014].
15. IETF RFC 5041. Direct data placement over reliable transports. Available from: http://www.ietf.org/rfc/rfc5041 [accessed on 27 November 2014].
16. IEEE 802.1. Data Center Bridging (DCB) task group. Available from: http://www.ieee802.org/1/pages/dcbridges.html [accessed on 27 November 2014].
17. Chen D, Eisley NA, Heidelberger P, Senger RM, Sugawara Y, Kumar S, Salapura V, Satterfield DL, Steinmacher-Burow B, Parker JJ. The IBM Blue Gene/Q interconnection network and message unit. *Proceedings of 23rd ACM/IEEE Supercomputing Conference (SC'11)*, Seattle, WA, USA, 2011; 26:1–26:10.
18. Alverson R, Roweth D, Kaplan L. The Gemini system interconnect. *Proceedings of 18th IEEE Annual Symposium on High-Performance Interconnects (HOTI'10)*, Google Campus, Mountain View, CA, USA, 2010; 83–87.
19. Kim J, Dally WJ, Scott S, Abts D. Technology-driven, highly-scalable Dragonfly topology. *Proceedings of 35th Annual International Symposium on Computer Architecture (ISCA'08)*, Beijing, China, 2008; 77–88.
20. The OpenFabrics Alliance (OFA). Available from: http://www.openfabrics.org/ [accessed on 27 November 2014].
21. IETF RFC 4392. IP over InfiniBand (IPoIB) architecture. Available from: http://www.ietf.org/rfc/rfc4392 [accessed on 27 November 2014].
22. Goldenberg D, Kagan M, Ravid R, Tsirkin MS. Zero copy sockets direct protocol over InfiniBand - preliminary implementation and performance analysis. *Proceedings of 13th IEEE Annual Symposium on High-Performance Interconnects (HOTI'05)*, Stanford University, CA, USA, 2005; 128–137.
23. Fox G, Furmanski W. Java for parallel computing and as a general language for scientific and engineering simulation and modeling. *Concurrency: Practice and Experience* 1997; **9**(6):415–425.
24. Thiruvathukal GK, Dickens PM, Bhatti S. Java on networks of workstations (JavaNOW): a parallel computing framework inspired by Linda and the Message Passing Interface (MPI). *Concurrency: Practice and Experience* 2000; **12**(11):1093–1116.
25. Dunning D, Regnier G, McAlpine G, Cameron D, Shubert B, Berry F, Merritt AM, Gronke E, Dodd C. The virtual interface architecture. *IEEE Micro* 1998; **18**(2):66–76.
26. Chang C-C, von Eicken T. Javia: a Java interface to the virtual interface architecture. *Concurrency: Practice and Experience* 2000; **12**(7):573–593.
27. Welsh M, Culler D. Jaguar: enabling efficient communication and I/O in Java. *Concurrency: Practice and Experience* 2000; **12**(7):519–538.
28. van Nieuwpoort RV, Maassen J, Wrzesinska G, Hofman R, Jacobs C, Kielmann T, Bal HE. Ibis: a flexible and efficient Java-based grid programming environment. *Concurrency and Computation: Practice and Experience* 2005; **17**(7–8):1079–1107.
29. Righi RR, Navaux POA, Cera MC, Pasin M. Asynchronous communication in Java over InfiniBand and DECK. *Proceedings of 17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'05)*, Rio de Janeiro, Brazil, 2005; 176–183.
30. Taboada GL, Touriño J, Doallo R. Java Fast Sockets: enabling high-speed Java communications on high performance clusters. *Computer Communications* 2008; **31**(17):4049–4059.
31. Baduel L, Baude F, Caromel D. Object-oriented SPMD. *Proceedings of 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'05)*, Cardiff, Wales, UK, 2005; 824–831.
32. Philippsen M, Haumacher B, Nester C. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience* 1999; **12**(7):495–518.
33. Maassen J, Van Nieuwpoort RV, Veldema R, Bal H, Kielmann T, Jacobs C, Hofman R. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems* 2001; **23**(6):747–775.
34. Taboada GL, Teijeiro C, Touriño J. High performance Java remote method invocation for parallel computing on clusters. *Proceedings of 12th IEEE Symposium on Computers and Communications (ISCC'07)*, Aveiro, Portugal, 2007; 233–239.

DOI: 10.1002/cpe

35. Lobosco M, Silva A, Loques O, de Amorim CL. A new distributed JVM for cluster computing. *Proceedings of 9th European Conference on Parallel Processing (Euro-Par'03)*, Klagenfurt, Austria, 2003; 1207–1215.
36. Veldema R, Hofman RF, Bhoedjang RAF, Bal HE. Run-time optimizations for a Java DSM implementation. *Concurrency and Computation: Practice and Experience* 2003; **15**(3-5):299–316.
37. Veldema R, Philippsen M. Evaluation of RDMA opportunities in an object-oriented DSM. *Proceedings of 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC'07)*, Urbana, IL, USA, 2007; 217–231.
38. Huang W, Zhang H, He J, Han J, Zhang L. Jdib: Java applications interface to unshackle the communication capabilities of InfiniBand networks. *Proceedings of 4th IFIP International Conference on Network and Parallel Computing Workshops (NPC'07)*, Dalian, China, 2007; 596–601.
39. Huang W, Han J, He J, Zhang L, Lin Y. Enabling RDMA capability of InfiniBand network for Java applications. *Proceedings of 4th IEEE International Conference on Networking, Architecture, and Storage (NAS'08)*, Chongqing, China, 2008; 187–188.
40. Stuedi P, Metzler B, Trivedi A. jVerbs: ultra-low latency for data center applications. *Proceedings of 4th ACM Annual Symposium on Cloud Computing (SOCC'13)*, Santa Clara, CA, USA, 2013; 10:1–10:14.
41. Baker M, Carpenter B, Fox G, Ko SH, Lim S. mpiJava: an object-oriented Java interface to MPI. *Proceedings of 1st International Workshop on Java for Parallel and Distributed Computing (IWJPDC'99)*, San Juan, Puerto Rico, 1999; 748–762.
42. Carpenter B, Fox G, Ko SH, Lim S. mpiJava 1.2: API specification. Available from: http://www.hpjava.org/reports/mpiJava-spec/mpiJava-spec/mpiJava-spec.html [accessed on 27 November 2014].
43. Taboada GL, Touriño J, Doallo R, Shafi A, Baker M, Carpenter B. Device level communication libraries for high-performance computing in Java. *Concurrency and Computation: Practice and Experience* 2011; **23**(18):2382–2403.
44. Gabriel E, Fagg GE, Bosilca G, Angskun T, Dongarra JJ, Squyres JM, Sahay V, Kambadur P, Barrett B, Lumsdaine A, Castain RH, Daniel DJ, Graham RL, Woodall TS. Open MPI: goals, concept, and design of a next generation MPI implementation. *Proceedings of 11th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'04)*, Budapest, Hungary, 2004; 97–104.
45. Cheptsov A. Promoting data-centric supercomputing to the WWW world: Open MPI's Java bindings. *Proceedings of 1st International Workshop on Engineering Object-Oriented Parallel Software (EOOPS'13)*, Barcelona, Spain, 2013; 1417–1422.
46. Baker M, Carpenter B, Shafi A. MPJ Express: towards thread safe Java HPC. *Proceedings of 8th IEEE International Conference on Cluster Computing (CLUSTER'06)*, Barcelona, Spain, 2006; 1–10.
47. Myrinet eXpress (MX). A high performance, low-level, message-passing interface for Myrinet. Version 1.2., October 2006. Available from: http://www.myricom.com/scs/MX/doc/mx.pdf [accessed on 27 November 2014].
48. Qamar B, Javed A, Jameel M, Shafi A, Carpenter B. Design and implementation of hybrid and native communication devices for Java HPC. *Proceedings of International Conference on Computational Science (ICCS'14)*, Cairns, Australia, 2014; 184–197.
49. Liu J, Wu J, Kini SP, Wyckoff P, Panda DK. High performance RDMA-based MPI implementation over InfiniBand. *Proceedings of 17th International Conference on Supercomputing (ICS'03)*, San Francisco, CA, USA, 2003; 295–304.
50. Liu J, Jiang W, Wyckoff P, Panda DK, Ashton D, Buntinas D, Gropp WD, Toonen BR. Design and implementation of MPICH2 over InfiniBand with RDMA support. *Proceedings of 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, Santa Fe, NM, USA, 2004; 24:1–24:10.
51. MPICH: a high performance and widely portable implementation of the MPI standard. Available from: http://www.mpich.org/ [accessed on 27 November 2014].
52. Sur S, Jin H-W, Chai L, Panda DK. RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits. *Proceedings of 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'06)*, New York City, NY, USA, 2006; 32–39.
53. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. Available from: http://mvapich.cse.ohio-state.edu/ [accessed on 27 November 2014].
54. Graham RL, Brightwell R, Barrett B, Bosilca G, Pješivac-Grbović J. An evaluation of Open MPI's matching transport layer on the Cray XT. *Proceedings of 14th European PVM/MPI User's Group Meeting (EuroPVM/MPI'07)*, Paris, France, 2007; 161–169.
55. Pritchard H, Gorodetsky I, Buntinas D. A uGNI-based MPICH2 Nemesis network module for the Cray XE. *Proceedings of 18th European MPI Users' Group Meeting (EuroMPI'11)*, Santorini, Greece, 2011; 110–119.
56. Gutierrez SK, Hjelm NT, Venkata MG, Graham RL. Performance evaluation of Open MPI on Cray XE/XK systems. *Proceedings of 20th IEEE Annual Symposium on High-Performance Interconnects (HOTI'12)*, Santa Clara, CA, USA, 2012; 40–47.
57. Cray Inc. Using the GNI and DMAPP APIs, September 2013. Available from: http://docs.cray.com/books/S-2446-51/S-2446-51.pdf [accessed on 27 November 2014].
58. Karo M, Lagerström R, Kohnke M, Albing C. The application level placement scheduler. *Proceedings of 48th Cray User Group (CUG'06)*, Lugano, Switzerland, 2006; 1–7.
59. Balaji P, Buntinas D, Goodell D, Gropp W, Krishna J, Lusk E, Thakur R. PMI: a scalable parallel process-management interface for extreme-scale systems. *Proceedings of 17th European MPI Users' Group Meeting (EuroMPI'10)*, Stuttgart, Germany, 2010; 31–41.

DOI: 10.1002/cpe

60. Tezuka H, O'Carroll F, Hori A, Ishikawa Y. Pin-down cache: a virtual memory management technique for zero-copy communication. *Proceedings of 12th International Parallel Processing Symposium (IPPS'98)*, Orlando, FL, USA, 1998; 308–314.
61. Saini S, Ciotti R, Gunney BTN, Spelce TE, Koniges A, Dossa D, Adamidis P, Rabenseifner R, Tiyyagura SR, Mueller M, Fatoohi R. Performance evaluation of supercomputers using HPCC and IMB benchmarks. *Journal of Computer and System Sciences* 2008; **74**(6):965–982.
62. HLRS' Hermit supercomputer in the TOP500 list. Available from: http://www.top500.org/system/177473 [accessed on 27 November 2014].
63. Bailey DH, Barszcz E, Barton JT, Browning DS, Carter RL, Dagum L, Fatoohi RA, Frederickson PO, Lasinski TA, Schreiber RS, Simon HD, Venkatakrishnan V, Weeratunga SK. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications* 1991; **5**(3):63–73.
64. Mallón DA, Taboada GL, Touriño J, Doallo R. NPB-MPJ: NAS parallel benchmarks implementation for message-passing in Java. *Proceedings of 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP'09)*, Weimar, Germany, 2009; 181–190.
65. Shafi A, Hussain A, Raza J. A parallel implementation of the finite-domain time-difference algorithm using MPJ Express. *Proceedings of 10th International Workshop on Java and Components for Parallelism, Distribution and Concurrency (IWJPDC'08)*, Miami, FL, USA, 2008; 1–6.

DOI: 10.1002/cpe