*Manuscript
Click here to download Manuscript: fmpj.pdf
Click here to view linked References

Noname manuscript No.
(will be inserted by the editor)

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

FastMPJ: a Scalable and Efficient Java Message-Passing Library

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

Received: / Accepted:

Abstract The performance and scalability of communications are key for High Performance Computing (HPC) applications in the current multi-core era. Despite the significant benefits (e.g., productivity, portability, multithreading) of Java for parallel programming, its poor communications support has hindered its adoption in the HPC community. This paper presents FastMPJ, an efficient Message-Passing in Java (MPJ) library, boosting Java for HPC by: (1) providing high-performance shared memory communications using Java threads; (2) taking full advantage of high-speed cluster networks (e.g., InfiniBand) to provide low-latency and high bandwidth communications; (3) including a scalable collective library with topology aware primitives, automatically selected at runtime; (4) avoiding Java data buffering overheads through zero-copy protocols; and (5) implementing the most widely extended MPI-like Java bindings for a highly productive development. The comprehensive performance evaluation on representative testbeds (InfiniBand, 10 Gigabit Ethernet, Myrinet, and shared memory systems) has shown that FastMPJ communication primitives rival native MPI implementations, significantly improving the efficiency and scalability of Java HPC parallel applications.

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

S. Ramos
e-mail: sramos@udc.es

G.L. Taboada
e-mail: taboada@udc.es

J. Touriño
e-mail: juan@udc.es

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

Keywords High Performance Computing (HPC) · Parallel Computing · Message-Passing in Java (MPJ) · Communication Middleware · High-Speed Networks · Performance Evaluation

1 Introduction

Java is currently among the preferred programming languages in web-based and distributed computing environments, and is an attractive option for High Performance Computing (HPC) [36]. Java provides some interesting characteristics of special benefit for parallel programming: built-in multithreading and networking support in the core of the language, in addition to its other traditional advantages for general programming such as object orientation, automatic memory management, portability, easy-to-learn properties, an extensive API and a wide community of developers.

Although Java was severely criticized for its poor computational performance in its beginnings [13], the performance gap between Java and natively compiled languages (e.g., C/C++, Fortran) has been narrowing for the last years [33,36]. The Java Virtual Machine (JVM), which executes Java applications, is now equipped with Just-in-Time (JIT) compilers that can obtain native performance from Java bytecode [35]. Nevertheless, the significant improvement in its computational performance is not enough to be a successful language in the area of parallel computing, as the performance of the communications is also essential to achieve high scalability in Java for HPC, especially in the current multi-core era.

Message-Passing Interface (MPI) [27] is the most widely used parallel programming paradigm and it is highly portable, scalable and provides good performance. It is the preferred choice for writing parallel applications on distributed memory systems such as multicore clusters, currently the most popular system deployments thanks to their interesting cost/performance ratio. Here, Java represents an attractive alternative to natively compiled languages traditionally used in HPC, for the development of applications for these systems as it provides built-in networking and multithreading support, key features for taking full advantage of hybrid shared/distributed memory architectures. Thus, Java can resort to threads in shared memory (intra-node) and to its networking support for distributed memory (inter-node) communications.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

The increasing number of cores per system demands efficient and scalable message-passing communication middleware in order to meet the ever growing computational power needs. Moreover, current system deployments are aggregating a significant number of cores through advanced high-speed cluster networks such as InfiniBand (IB) [24], which usually provide interesting features such as Remote Direct Memory Access (RDMA) support, increasing the complexity of communication protocols. However, up to now Message-Passing in Java (MPJ) [17] implementations have been focused on providing new functionalities, rather than concentrate on developing efficient communications on high-speed networks and shared memory systems. This lack of efficient communication support in Java, especially in the presence of high-speed cluster networks, results in lower performance than native MPI implementations. 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 presents FastMPJ, our efficient and scalable MPJ implementation for parallel computing, which addresses all these issues. Thus, FastMPJ provides high-performance shared memory communications, efficient support of high-speed networks, as well as a scalable collective library which includes topology aware primitives. The comprehensive performance evaluation has shown that FastMPJ is competitive with native MPI libraries, which increases the scalability of communication-intensive Java HPC parallel applications.

The structure of this paper is as follows: Section 2 presents background information about MPJ. Section 3 introduces the related work. Section 4 describes the overall design of FastMPJ. Section 5 details some aspects of the FastMPJ implementation, including point-to-point and collective communications support. Comprehensive benchmarking results from FastMPJ evaluation are shown in Section 6. Finally, Section 7 summarizes our concluding remarks.

2 Message-Passing in Java

Soon after the introduction of Java, there have been several implementations of MPJ libraries. However, the MPI standard [27] defines bindings for C, C++ and Fortran programming languages. Therefore, as there are no bindings for the Java language in the standard, most of the initial MPJ projects have developed their own MPI-like bindings. In contrast, most recent projects generally adhere to one of the two major MPI-like Java bindings which have been proposed by the community: (1) the mpiJava 1.2 API [16], the most widely extended, which supports an MPI C++-like interface for the MPI 1.1 subset, and (2) the JGF MPJ API [17], which is the proposal of the Java Grande Forum (JGF) [1].

MPJ libraries are usually implemented in three ways: (1) using some high-level Java messaging API like Remote Method Invocation (RMI) to implement a "pure" Java message-passing system (i.e., 100% Java code); (2) wrapping an underlying native MPI library through the Java Native Interface (JNI); or (3) following a hybrid layered design, which includes a pluggable architecture based on an idea of low-level communication devices. Thus, hybrid libraries provide Java-based implementations of the high-level features of MPI at the top levels of the software. Hence, they can offer a "pure" Java approach through the use of Java-based communication devices (e.g., via Java sockets), and additionally a higher performance approach through low-level native communication devices that use JNI to take advantage of specialized HPC hardware. Although most of the Java communication middleware is based on RMI, MPJ libraries looking for efficient communication have followed the latter two approaches.

Generally, applications implemented on top of Java messaging systems can have different requirements. Thus, for some applications the main concern could be portability, while for others could be high-performance communications. Each of the above solutions fit with specific situations, but can present associated trade-offs. On the one hand, the use of RMI ensures portability, but it may not provide an efficient solution, especially in the presence of high-speed HPC hardware. On the other hand, the wrapper-based approach presents some inherent portability and instability issues derived from the native code, as these implementations have to wrap all the methods of the MPJ API. Moreover, the support of multiple heterogeneous runtime platforms, MPI libraries and JVMs entails a significant maintenance effort, although usually in exchange for higher performance than RMI. However, the hybrid approach minimizes the JNI code to the bare minimum using low-level pluggable communication devices, being the only solution that can ensure both requirements. Nevertheless, most of the MPJ projects that conform with this hybrid design rely on Java sockets and inefficient TCP/IP emulations to support current HPC communication hardware (e.g., InfiniBand). Although the use of Java sockets usually outperforms RMI-based middleware, it requires an important programming effort. Furthermore, the use of the sockets API in a communication device still represents an important source of overhead and lack of scalability in Java communications, especially in the presence of high-speed networks [23].

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

3 Related Work

Multiple MPI native implementations have been developed, improved and maintained over the last 15 years intended for cluster, grid and emerging cloud computing environments. Regarding MPJ libraries, there have been several efforts to develop a Java message-passing system for HPC since its introduction [36,40]. However, most of the developed projects over the last decade were prototype implementations, without maintenance. Currently, the most relevant ones in terms of uptake by the HPC community are mpiJava, MPJ Express, MPJ/Ibis, and FastMPJ, next presented.

mpiJava [9] is a Java message-passing system that consists of a collection of wrapper classes that use JNI to interact with an underlying native MPI library. This project implements the mpiJava 1.2 API and has been perhaps the most successful Java HPC messaging system, in terms of uptake by the community. However, mpiJava can incur a noticeable overhead, especially for large messages, and also presents some portability and instability issues. Thus, it only supports some native MPI implementations, as wrapping a wide number of methods and heterogeneous runtime platforms entails a significant maintenance effort, as mentioned before.

MPJ Express [11] is one of the projects that conforms with the aforementioned hybrid approach. This library implements the mpiJava 1.2 API and presents a modular design which includes a pluggable architecture of communication devices that allows to combine the portability of the "pure" Java New I/O (NIO) communications package together with the native Myrinet support through JNI. Additionally, it provides shared memory support using Java threads [34]. However, this project poses several important issues: (1) its overall design relies on a buffering layer [12] that significantly limits performance and scalability of communications; (2) it lacks efficient support for InfiniBand (IB), the most widely adopted networking technology in current HPC clusters; (3) it includes poorly scalable collective algorithms; and (4) its bootstrapping mechanism typically exhibits some issues in specific environments.

MPJ/Ibis [15] is another hybrid project that, in this case, conforms with the JGF MPJ API. Actually, this library is implemented on top of Ibis [29], a parallel and distributed Java computing framework. Thus, it can use either "pure" Java communications, based on Java sockets, or native communications on Myrinet. However, the Myrinet support is based on the GM library, an out-of-date low-level API which has been superseded by the MX (Myrinet Express) library [28]. Moreover, MPJ/Ibis also lacks efficient IB support, and additionally, does not provide efficient shared memory and collective communications. Furthermore, MPJ/Ibis does not fully implement some high-level features of MPI (e.g., inter-communicators and virtual topologies).

FastMPJ is our Java message-passing implementation of the mpiJava 1.2 API, which also presents a hybrid design approach. The initial prototype implementation was presented as a proof of concept in [38]. This prototype only implemented a small subset of the communications-related API. Furthermore, it only included one communication device implemented on top of Java IO sockets, which severely limited its overall scalability and performance. Although the use of high-performance socket implementations, such as the Java Fast Sockets (JFS) project [37], can improve performance on shared memory and high-speed networks, the use of sockets in a communication device can not provide an efficient and scalable solution, as mentioned in the previous section.

Currently, FastMPJ has overcome these limitations by: (1) implementing the remaining of the mpiJava 1.2 API (e.g., virtual topologies, inter-communicators and groups operations are currently available), except part of the derived data types (e.g., Vector, Struct) since Java can provide any user-defined structure natively, by using objects, which fits more straightforwardly into an object-oriented programming model; (2) providing high-performance shared memory support; (3) efficiently supporting high-speed cluster networks, especially IB; and (4) implementing a user friendly and scalable bootstrapping mechanism to start the Java parallel processes. The overcoming of the previous limitations of FastMPJ, together with the implementation of an efficient communications support which provides similar performance to native MPI libraries, are the main contributions of this paper.

Additionally, some previous works have already evaluated the aforementioned MPJ libraries [38,39]. As main conclusions, these studies have assessed that FastMPJ is the best performer among them, overcoming some of the previous performance limitations such as the high buffering penalty and the JNI overhead. Moreover, most of the MPJ projects, especially mpiJava and MPJ/Ibis, are currently outdated and without active development. Due to these drawbacks, mainly low performance and lack of up-to-date development, the performance evaluation carried out in Section 6 only considers the comparison of FastMPJ against native MPI libraries, for clarity purposes.

Finally, there have also been some additional works that focused on other important aspects of Java to be a successful option in HPC, such as providing high-performance file I/O [14,19].

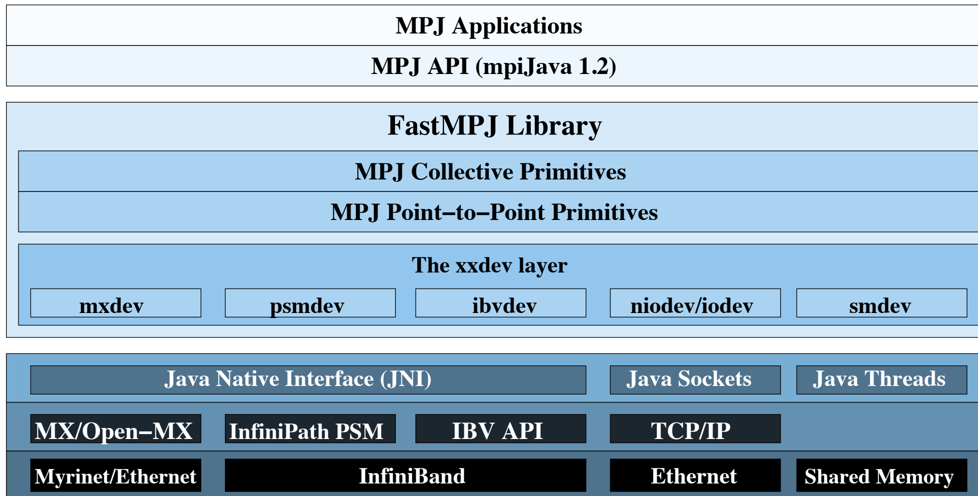

Fig. 1 Overview of the FastMPJ layered design

4 FastMPJ Design

Figure 1 presents an overview of the FastMPJ layered design and the different levels of the software. The MPJ communications API, which includes both collective and point-to-point primitives, is implemented on top of the `xxdev` device layer. The device layer has been designed as a simple and pluggable architecture of low-level communication devices. Moreover, this layer supports the direct communication of any serializable Java object without data buffering, whereas `xdev` [10], the API that `xxdev` is extending, does not support this direct communication. Thus, the `xdev` API, which is used internally by the MPJ Express library, relies on a buffering layer [12] which is only able to transfer the custom `xdev` buffer objects. This fact adds a noticeable copying overhead [36], especially for large messages, which prevents MPJ Express to implement zero-copy protocols. The avoidance of this intermediate data buffering overhead on the critical path of communications is the main benefit of the `xxdev` device layer with respect to its predecessor. Thus, this fact allows `xxdev` communication devices to implement zero-copy protocols when communicating primitive data types using, for instance, RDMA-capable high-speed cluster networks. Additional benefits of this API are its flexibility, portability and modularity thanks to its encapsulated design.

In more detail, the `xxdev` layer provides a Java low-level message-passing API (see Listing 1) with basic operations such as point-to-point blocking (`send` and `recv`) and non-blocking (`isend` and `irecv`) communication methods. Moreover, it also includes synchronous communications (`ssend` and `issend`) and functions to check incoming messages without actually receiving them (`probe` and `iprobe`). Thus, an `xxdev` device is similar to an MPI communicator, but with reduced functionality. This simple design eases significantly the development of `xxdev` communications devices in order to provide custom support of high-speed cluster networks (e.g., High-speed Ethernet and IB) and shared memory systems, while leveraging other infrastructure provided by the upper levels of FastMPJ, such as the runtime system and the layer that provides the full MPJ semantics (e.g., virtual topologies, inter-communicators). With this modular design FastMPJ enables its incremental development and provides the capability to update and swap layers in or out as required. Thus, end users can opt at runtime to use a high-performance native network device, or choose a “pure” Java device, based either on sockets or threads, for portability.

5 FastMPJ Implementation

FastMPJ communication support relies on the efficient implementation of low-level `xxdev` devices on top of specific native libraries and HPC communication hardware. Currently, FastMPJ includes three communication devices that support high-speed cluster networks: (1) `mxdev`, for Myrinet and High-speed Ethernet; (2) `psmdev`, for Intel/QLogic InfiniBand adapters; and (3) `ibvdev`, for InfiniBand adapters in general terms. These devices are implemented on top of MX/Open-MX, InfiniPath PSM and IB Verbs (IBV) native libraries, respectively (see Figure 1). Although these underlying native libraries have been initially designed for inter-node network-based communication, in the particular case of MX/Open-MX and PSM also provide efficient

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

  abstract Request isend(Object msg,PID dst,int tag,int context);
  abstract Request irecv(Object msg,PID src,int tag,int context,Status status);
  abstract void send(Object msg,PID dst,int tag,int context);
  abstract Status recv(Objecct msg,PID src,int tag,int context);
  abstract Request issend(Object msg,PID dst,int tag,int context);
  abstract void ssend(Object msg,PID src,int tag,int context);

  abstract Status iprobe(PID src,int tag,int context);
  abstract Status probe(PID src,int tag,int context);
}
```

Listing 1 xxdev API

intra-node shared memory communication, usually implemented through some Inter-Process Communication (IPC) mechanism. Thus, this fact allows FastMPJ to take full advantage of hybrid shared/distributed memory architectures, such as clusters of multi-core nodes, except for the `ibvdev` device, as IBV does not support shared memory. Additionally, the TCP/IP stack (`niodev` and `iodev`) and high-performance shared memory systems (`smdev`) are also supported through "pure" Java communication devices, which ensures portability.

The user-level methods of the MPJ API related to the collective and point-to-point communication layers are implemented on top of these `xxdev` communication devices. This may involve some native code depending on the underlying device being used (e.g., `ibvdev` and `psmdev` for IB support). The rest of the high-level abstractions of the MPJ API (e.g., virtual topologies, intra- and inter-communicators, groups operations) is implemented in "pure" Java code (i.e., 100% Java). Hence, this implementation can ensure both portability and/or high-performance requirements of Java message-passing applications, while avoiding some of the associated problems of the wrapper-based approach through JNI, as mentioned in Section 2 (e.g., instability and portability issues, high maintenance effort). These issues are derived from the amount of native code that is involved using a wrapper-based implementation (note that all the methods of the MPJ API have to be wrapped). However, FastMPJ can minimize to the bare minimum the amount of JNI code needed to support a specific network device, as the `xxdev` devices only have to implement a very small number of methods (see Listing 1). In the next sections, the implementation of the various MPI features in FastMPJ will be discussed.

5.1 High-Speed Networks Support

FastMPJ provides efficient support for high-speed cluster networks through `mxdev`, `ibvdev` and `psmdev` communications devices, next presented.

5.1.1 Myrinet/High-speed Ethernet

The `mxdev` device implements the `xxdev` API on top of the Myrinet Express (MX) library [28], which runs natively on Myrinet networks. More recently, the MX API has also been supported in high-speed Ethernet networks (10/40 Gigabit Ethernet), both on Myricom specialized NICs and on any generic Ethernet hardware through the Open-MX [22] open-source project. Thus, the TCP/IP stack can be replaced by `mxdev` transfers over Ethernet networks providing higher performance than using standard Java sockets. Moreover, the `mxdev` device can also take advantage of the efficient intra-node shared memory communication protocol implemented by MX/Open-MX [21] to improve the performance of networked applications in multi-core systems.

In MX messages are exchanged among endpoints, which are software representations of Myrinet/Ethernet NICs. Every message operation, either sending or receiving, starts with a non-blocking communication request (e.g., mx_isend), which is queued by MX, returning the control to `mxdev`. Then, the `mxdev` device is responsible for checking the successful completion of the communication operation. The message matching mechanism at the receiver side is based on a 64-bit matching field, specified by both communication peers, in order to deliver incoming messages to the right receive requests.

The MX API is only available in C, thus the `mxdev` device implements `xxdev` methods calling MX functions

through JNI. Moreover, as MX already provides a low-level messaging API which closely matches the `xxdev` layer, `mxdev` deals with the Java objects marshaling and communication, the JNI transfers and the MX parameters handling. Therefore, FastMPJ with `mxdev` provides the user with a higher level messaging API than MX, also freeing Java developers from the implementation of JNI calls, which benefits programmability without trading off much performance.

5.1.2 InfiniBand

The native and efficient InfiniBand (IB) support is also included in FastMPJ with `ibvdev` and `psmdev` devices. On the one hand, the `ibvdev` device directly implements its own communication protocols through JNI on top of the IBV API, which is part of the OpenFabrics Enterprise Distribution (OFED [30]), an open-source software for RDMA and kernel bypass applications. The native support of the IBV API in Java is somewhat restricted so far to native MPI libraries, as previous MPJ libraries relied on the TCP/IP emulation over IB protocol (IPoIB) [25], which provides significantly poorer performance, especially for short messages [23].

A previous implementation of the `ibvdev` device was firstly integrated into the MPJ Express library [20] as a proof of concept, but only for internal testing purposes as it was never part of the official release. Although it was able to provide higher performance than using the IPoIB protocol, the buffering layer in MPJ Express significantly limited its performance and scalability. Therefore, the `ibvdev` device had to be reimplemented to conform with the `xxdev` API and then adapted for its integration into the FastMPJ library in order to improve its performance. Thus, FastMPJ achieves start-up latencies and bandwidths similar to native MPI performance results on IB networks thanks to the efficient, lightweight and scalable communication protocol implemented in `ibvdev`, which includes a zero-copy mechanism for large messages using the RDMA-write operation.

On the other hand, another original contribution of this paper is the introduction of the `psmdev` device, which provides for the first time in Java native support for the InfiniPath family of Intel/QLogic IB adapters over the Performance Scaled Messaging (PSM) interface. PSM is a low-level user-space messaging library which implements an intra-node shared memory and inter-node communication protocol, which are completely transparent to the application.

In order to establish the initial connections between endpoints, the `psmdev` device has to rely on an out-of-band mechanism, which has been implemented with TCP sockets, to distribute the endpoint identifiers. After initializing endpoints, a Matched Queue (MQ) interface is created and can be used to send and receive messages. The MQ interface semantics are consistent with those defined by the MPI 1.2 standard for message-passing between two processes. Thus, incoming messages are stored according to their tags to preposted receive buffers. The PSM API is only available in C; thus, following a similar approach to `mxdev`, the `psmdev` device also implements `xxdev` methods calling PSM functions through JNI dealing with the Java Objects marshaling and communication, the JNI transfers and the PSM parameters handling. Although the Intel/QLogic adapters are also supported by the `ibvdev` device through the IBV API, `psmdev` usually achieves significantly higher performance than using `ibvdev`, as PSM is specifically designed and highly tuned by Intel/QLogic for its own IB adapters.

5.2 Socket-based Communications Devices

Initially, FastMPJ included only one communication device implemented on top of Java IO sockets (`iodev`), which turned out to be the limiting factor in performance and scalability, especially for non-blocking communication. This fact has motivated the implementation of a new communication device based on Java NIO sockets (`niodev`), which include more scalable non-blocking communication support by providing `select()` like functionality. Additionally, a new socket-based device (`sctpdev`) implemented on top of Stream Control Transmission Protocol (SCTP) sockets is currently work in progress.

Nevertheless, these “pure” Java communication devices are only provided for portability reasons, as they rely on the ubiquitous TCP/IP stack, which introduces high communication overhead and limited scalability for communication-intensive HPC applications.

5.3 Shared Memory Communications

FastMPJ includes a “pure” Java thread-based communication device (`smdev`) that efficiently supports shared memory intra-node communication [31], thus being able to exploit the underlying multi-core architecture replacing inter-process and network-based communications by Java threads and shared memory intra-process transfers.

In this thread-based device, there is a single JVM instance and each MPJ rank in the parallel application (i.e., each Java process in the case of using a network-based communication device) is represented by a Java

thread. Consequently, message-passing communication between these threads is achieved using shared data structures. Therefore, the FastMPJ runtime must create a single JVM with as many Java threads as the number of ranks exist in the global communicator (i.e., MPI.COMM_WORLD), which depends on an input parameter that is specified by the user when starting the MPJ application.

An obvious advantage of this approach, especially in the context of Java, is that an application does not compromise portability. Moreover, the use of a single JVM can take advantage of lower memory consumption and garbage collection overhead. Furthermore, while multithreading programming allows to exploit shared memory intra-process transfers, it usually increases the development complexity due to the need for thread control and management, task scheduling, synchronization, and maintenance of shared data structures. Thus, using the `smdev` device, the developer does not have to deal with the issues of the multithreading programming, as `smdev` offers a high level of abstraction that supports handling threads as message-passing processes, providing similar or even higher performance than native MPI implementations.

5.3.1 Class Loading

The use of threads in the `smdev` device requires the isolation of the namespace for each thread, configuring a distributed memory space in which they can exchange messages through shared memory references. While processes from different JVMs are completely independent entities, threads within a JVM are instances of the same application class, sharing all static variables. Thus, this device creates each running thread with its custom class loader. Therefore, all the non-shared classes within a thread can be directly isolated in its own namespace in order to behave like independent processes. Nevertheless, communication through shared memory transfers requires the access to several shared classes within the device. When the system loader does not find a class, the custom class loader is used, following the JVM class loader hierarchy. This mechanism implies that the system class loader loads every reachable class that, in consequence, is shared by all threads. Thus, its classpath has to be bounded in such a way that it only has access to shared packages that contain the implementation of shared memory transfers among threads. Consequently, communications are delegated to a shared class which allocates and manages shared message queues (a pair of queues per thread) in order to implement the data transfers as regular data copies between threads, thus providing a highly efficient zero-copy protocol.

Finally, the use of a pair of queues per thread enables `smdev` to include fine-grained synchronizations, combining busy waits and locks, thus reducing contention in the access to the shared structures. As an example, MPJ Express shared memory support [34] uses a global pair of queues with class lock-based synchronization, which can result in a very inefficient approach in applications that involve a high number of threads.

5.4 Scalable Collective Communications

The MPI specification defines collective communication operations as a convenience to application developers, which can save significant time in the development of parallel applications. FastMPJ provides a scalable and efficient collective communication library for parallel computing on multi-core architectures. This library includes topology aware primitives which are implemented on top of point-to-point communications, taking advantage of communications overlapping and obtaining significant performance benefits in collective-based communication-intensive MPJ applications. The library implements up to six algorithms per collective primitive, whereas previous MPJ libraries are usually restricted to one algorithm. Furthermore, the algorithms can be selected automatically at runtime, depending on the number of cores and the message length involved in the collective operation.

The collective algorithms present in the FastMPJ collective library can be classified in six types, namely Flat Tree (FT) or linear, Minimum-Spanning Tree (MST), Binomial Tree (BT), Four-ary Tree (FaT), Bucket (BKT) or cyclic, and BiDirectional Exchange (BDE) or recursive doubling, which have been extensively described in the literature [18].

5.5 Runtime System

Although the runtime system is not part of the MPI specification, it is an essential element which allows to execute processes across various platforms. Thus, the FastMPJ runtime system is in charge of starting the parallel Java processes across multiple machines, supporting several OSs either UNIX-based (e.g. GNU/Linux, MAC OS X) or Microsoft Windows-based (XP/Vista/7/8). In addition, the runtime does not assume a shared file system and it allows to run MPJ applications using both class and JAR file formats.

This fully portable runtime system mainly consists of two modules: (1) an fmpjd module (Java daemon

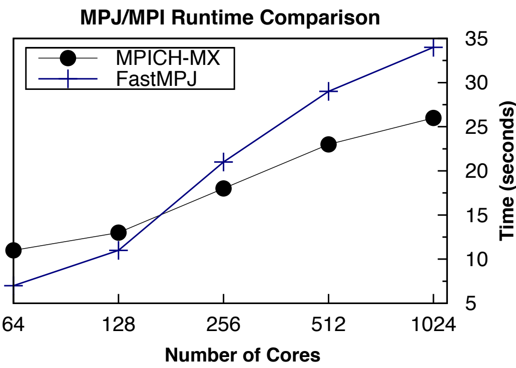

Fig. 2 MPJ/MPI deployment and initialization

listening on a configurable TCP port) which executes on compute nodes and listens for requests to start Java processes in a new JVM; and (2) an fmpjrun module, client of the Java daemons. In UNIX-based OSs, a set of Java daemons can be started/stopped over the network using SSH within the fmpjrun application, as the OS is automatically detected by FastMPJ. Moreover, the runtime system is also compatible with traditional job schedulers such as SGE/OGE, SLURM, LSF and PBS. Additionally, other modules are provided to start, stop and trace the status (running/not running) of the daemons. However, on Windows platforms, the daemons either need to be: (1) manually started, (2) configured to start automatically on OS startup, or (3) installed as a native service, as SSH utilities are not usually available in these platforms.

The FastMPJ runtime efficiently supports the handling of a high number of machines and processes. For instance, a 1024-core “Hello World” MPJ program can be executed in less than 35 seconds, including the time needed for starting the Java daemons and the initialization of the parallel environment. Figure 2 compares FastMPJ against MPICH-MX, which was configured with the SLURM PMI process launcher, running a “Hello World” example application on the MareNostrum testbed (see Table 1 in Section 6.1 for more details on this testbed).

6 Performance Evaluation

This section presents a comprehensive performance evaluation of the FastMPJ library compared to representative native MPI libraries: Open MPI [5], MVAPICH2 [3] and MPICH-MX [6], from point-to-point and collective message-passing primitives to the assessment of their impact on the scalability of representative parallel codes, using the NASA Advanced Supercomputing (NAS) Parallel Benchmarks suite (NPB) [4,8]. The NPB parallel codes have been selected as it is the benchmarking suite most commonly used in the evaluation of languages, libraries and middleware for HPC.

As mentioned in Section 3, previous works [38,39] have already characterized the performance of the other popular MPJ implementations (mpiJava, MPJ/Ibis and MPJ Express) against native MPI libraries, so for clarity purposes these MPJ implementations have not been re-evaluated. In fact, these libraries obtained poor performance, as shown in the references, and they have not been updated since their last evaluations.

6.1 Experimental Configuration

Table 1 shows the main characteristics of the five representative systems used in the performance evaluation. Both FastMPJ and native MPI libraries have been configured with the most efficient settings and communication device for each testbed (e.g., using only the shared memory device in shared memory systems).

Regarding distributed memory systems, the first testbed (from now on IB-QDR) is a multi-core cluster [7] that consists of 64 nodes, each of them with 24 GBytes of memory and 2 Intel Xeon quad-core Westmere-EP processors (hence 8 cores per node) interconnected via IB QDR (Mellanox-based NICs). The performance results for the collective primitives micro-benchmarking and the NPB kernels evaluation on this system have been obtained using 8 processes per node (hence 512 cores in total). The second system (from now on IB-DDR) is a multi-core cluster that consists of 16 nodes, each of them with 16 GBytes of memory and 2 Intel Xeon quad-core Nehalem-EP processors (hence 8 cores per node) interconnected via IB DDR (QLogic-based NICs). Additionally, two nodes have also one 10 Gigabit Ethernet (GbE) Intel NIC. Performance results on this testbed have also been obtained using 8 processes per node (hence 128 cores in total). The third system is the MareNostrum supercomputer [2] (from now on MN), which was ranked #465 in the TOP500 [41] list (June 2012). This supercomputer consists of 2560 nodes, each of them with 8 GBytes of memory and 2 PowerPC dual-core processors (hence 4 cores per node) interconnected via Myrinet 2000. General user accounts on this supercomputer are limited to use up to 1024 cores. Thus, performance results on this system have been obtained using 256 nodes and 4 processes per node (hence 1024 cores in total).

Regarding shared memory systems, the Intel-SHM testbed has 4 Intel Xeon ten-core Westmere-EX processors (hence 40 cores) and 512 GBytes of memory, whereas the AMD-SHM testbed provides with 4 AMD Opteron twelve-core Magny-Cours processors (hence 48

Table 1 Description of the systems used in the performance evaluation

	#nodes	CPU	Memory	#cores	NIC (Driver)	Network	OS (Kernel)	MPI libraries	JVM
IB-QDR	64	2 x 4-core Intel Xeon E5620	24 GBytes	512	Mellanox MT26428 (OFED 1.3)	IB QDR (32 Gbps)	CentOS (2.6.32)	Open MPI 1.4.4 MVAPICH2 1.6	OracleJDK 1.6.0_27
IB-DDR	16	2 x 4-core Intel Xeon E5520	16 GBytes	128	QLogic QLE7240 (OFED 1.5) Intel 82598EB (Open-MX 1.5.1)	IB DDR (16 Gbps) 10 GbE (10 Gbps)	CentOS (2.6.18)	Open MPI 1.4.5 MVAPICH2 1.7	OracleJDK 1.6.0_23
MN	2560	2 x 2-core IBM PowerPC 970MP	8 GBytes	10240	Myrinet 2000 (MX 1.2.7)	Myrinet (2 Gbps)	Suse (2.6.16)	MPICH-MX 1.2.7	IBM 1.7.0
Intel-SHM	1	4 x 10-core Intel Xeon E7 4850	512 GBytes	40	-	-	Ubuntu (3.2.0)	Open MPI 1.4.5 MVAPICH2 1.7	OpenJDK 1.6.0_23
AMD-SHM	1	4 x 12-core AMD Opteron 6172	128 GBytes	48	-	-	CentOS (2.6.32)	Open MPI 1.4.4 MVAPICH2 1.6	OracleJDK 1.6.0_23

cores) and 128 GBytes of memory. The NPB performance results on these systems have been executed using 1, 2, 4, 8, 16 and 32 cores. Thus, the maximum number of available cores in each shared memory system could not be used, as the selected NBP kernels only work for a number of cores which is a power of two.

The evaluation of message-passing communication primitives (Sections 6.2 and 6.3) has been carried out using a representative micro-benchmarking suite, the Intel MPI Benchmarks (IMB) [32], and our own MPJ counterpart, which adheres to the IMB measurement methodology. 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. In addition, these benchmark suites have been used without cache invalidaton, as it is more representative of a real scenario, where data to be transmitted is generally in cache.

Finally, the evaluation of representative message-passing parallel codes (Section 6.4) has used the MPI and OpenMP implementations of the NPB suite (NPB-MPI/NPB-OMP version 3.3) together with its MPJ counterpart (NPB-MPJ) [26]. Four representative NPB kernels have been evaluated: Conjugate Gradient (CG), Fourier Transform (FT), Integer Sort (IS) and Multi-Grid (MG), selected as they present medium to high communication intensiveness. The performance of two different common scaling metrics has been analyzed: (1) strong scaling (i.e., fix the problem size and vary the number of cores); and (2) weak scaling (i.e., vary the problem size linearly with the number of cores).

6.2 Point-to-point Micro-benchmarking

Figure 3 presents point-to-point performance results obtained on IB (top graphs), 10 GbE and Myrinet (middle graphs), and on shared memory systems (bottom graphs). The metric shown is the half of the round-trip time of a pingpong test for messages up to 1 KByte (left part of the graphs), and the bandwidth for messages larger than 1 KByte (right part).

On the IB-QDR testbed (top left graph), FastMPJ `ibvdev` device obtains 2.2 μs start-up latency, quite close to MPI results (around 1.9 μs). Regarding bandwidth results, `ibvdev` bandwidth is slightly lower than the MPI performance up to 64-KByte messages. From this point, `ibvdev` changes to an RDMA Write-based zero-copy protocol which is able to obtain similar bandwidths (up to 22.5 Gbps) to MPI libraries for large messages. On the IB-DDR testbed (top right graph), the `psmdev` device and Open MPI obtain the lowest start-up latency, around 1.9 μs, slightly outperforming MVAPICH2 (2 μs). The observed bandwidths are identical up to 128 KBytes, when MVAPICH2 gets slightly better results than Open MPI and FastMPJ in the message range [256 KBytes-2 MBytes]. For messages $\geq$ 2 MBytes, `psmdev` obtains up to 11.5 Gbps whereas MPI libraries only achieve a 6% more bandwidth, around 12.2 Gbps. These results confirm that `ibvdev` and `psmdev` devices implement highly efficient and lightweight communication protocols, which allows Java applications to take full advantage of the low-latency and high throughput provided by IB.

Regarding the 10 GbE testbed (middle left graph), `mxdev` gets start-up latencies as low as 15.6 μs, quite competitive compared to MPI libraries which obtain 11.2 μs and 11.5 μs for MVAPICH2 and Open MPI, respectively. Fortunately, this small gap disappears from 1 KByte, when `mxdev` and MVAPICH2 achieve identical bandwidths, whereas Open MPI results are the worst up to 2 MBytes. From this point, the network turns out to be the main performance bottleneck, as the maximum bandwidth achieved is around 9.4 Gbps for all evaluated libraries, quite close to the 10 Gbps limit for this networking technology. Here, the avoidance of the TCP/IP protocol is key for FastMPJ to obtain competitive results compared to MPI, especially for short messages, as the use of a socket-based device (`iodev` or `niodev`) would incur a significant overhead due to the poor performance of Java sockets. The results on the MN supercomputer over a Myrinet network (middle right graph) show that `mxdev` start-up latency gets even closer to MPI results, obtaining 5.2 and 4.1 μs, respectively. Their observed bandwidths are quite similar from 1 KByte, suffering the 2 Gbps limit for this networking technology.

Regarding shared memory systems, the performance results of the `smdev` device on the Intel-SHM testbed (bottom left graph) show even below 1 μs start-up latencies, but approximately twice the latency obtained by MPI libraries (around 0.42-0.48 μs). However, for message sizes $>$ 2 KBytes, the zero-copy thread-based intra-process protocol implemented by `smdev`, which allows direct data transfers between Java threads, clearly outperforms MPI libraries. Here, MPI libraries usually implement one-copy protocols since data transfers are inter-process communications through an intermediate shared memory structure, using IPC resources, which requires at least two data transfers. However, the direct communication in `smdev` does not show significant benefits in the latency of very short messages, as MPI libraries achieve lower start-up latencies for message sizes $<$ 2 KBytes. Thus, the thread synchronization overhead for `smdev`, which combines busy waits and locks, seems to be higher than the process synchronization

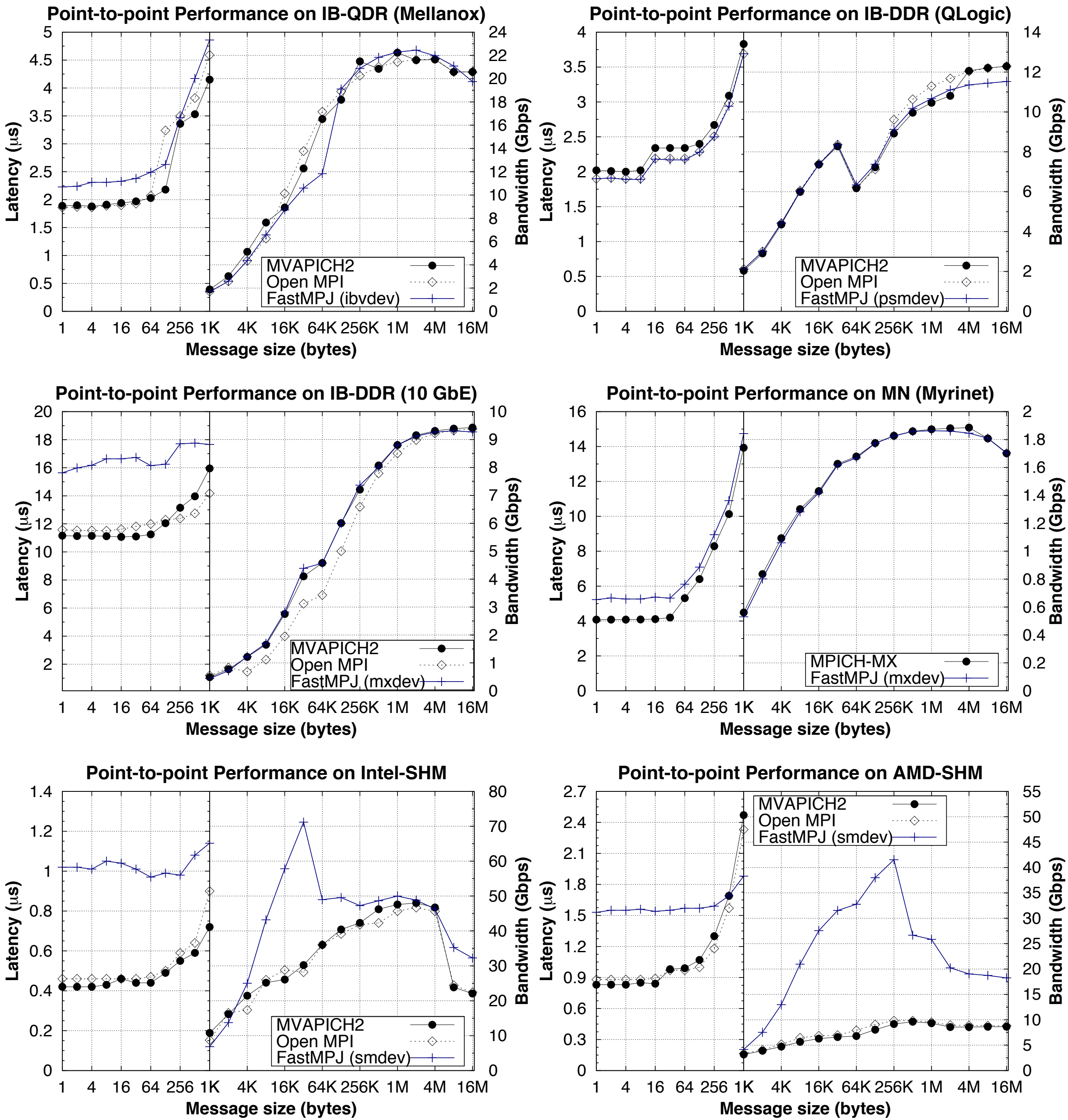

Fig. 3 Point-to-point performance on InfiniBand QDR and DDR, 10 Gigabit Ethernet, Myrinet and shared memory

overhead for MPI libraries, which usually use only lock-free algorithms. In addition, the high start-up latency overhead imposed by the JVM in the initialization of the copy is higher than the cost of the IPC extra copy performed by MPI when transferring short messages. As the overhead per byte transferred in MPI, which uses two data transfers, is higher than the combined overhead for `smdev` (thread synchronization plus JVM start-up latency), the consequence is that up to a certain threshold point (message size $<$ 2 KBytes), short messages have less overhead for MPI, whereas FastMPJ is the best performer for medium and large messages due to the avoidance of extra copies in `smdev`. Moreover, the `smdev` device obtains the highest performance (up to 71.2 Gbps) especially when messages are around the L1 cache size (32 KBytes). When the message does not fit in the L2 cache (256 KBytes), the performance gap between `smdev` and MPI reduces, which evidences

the impact of the memory hierarchy on shared memory performance, as no cache invalidation is performed in this test, as mentioned before.

The performance results on the AMD-SHM testbed (bottom right graph) show a similar pattern. Thus, MPI obtains lower start-up latencies than smdev, 0.88 μs and 1.53 μs, respectively, but relatively high compared to the Intel-SHM ones owing to the lower computational power of the AMD processor core. Regarding large message performance, smdev again clearly outperforms MPI libraries, obtaining up to 41.6 Gbps whereas MPI does not even reach 10 Gbps. This poor performance is explained by the low memory access throughput and the high copy penalty in this system. In addition, the peak bandwidth for smdev now is obtained for 256 KBytes (the L2 cache size in this system), not taking advantage of the messages fitting in the L1 cache (64 KBytes), while in the Intel testbed the peak was for 32 KBytes (the L1 cache size for this system).

The observed point-to-point communication efficiency of xxdev devices allows FastMPJ to provide low-latency and high-bandwidth communications for MPJ parallel applications, both on high-speed networks and high-performance shared memory systems. Furthermore, the obtained results are quite close to native MPI results, even outperforming them in some scenarios (e.g., large message performance in shared memory).

6.3 Collective Primitives Micro-benchmarking

Figure 4 presents the aggregated bandwidth for the broadcast primitive, a representative data movement operation, on the IB-QDR, IB-DDR, MN and AMD-SHM testbeds using all the available cores in each system. The aggregated bandwidth metric has been selected as it takes into account the global amount of data transferred (i.e., $message\ size * number\ of\ processes$).

On the IB-QDR testbed (top left graph), the ibvdev device obtains higher bandwidth than MVAPICH2 in the message range [2 KBytes - 256 KBytes]. However, Open MPI is the best performer, especially from 256 KBytes on. From this point, Open MPI dramatically increases its performance, which suggests that it switchs to a highly efficient algorithm for large messages (the same behaviour has been observed in the remaining scenarios where Open MPI is also evaluated). The IB-DDR testbed results (top right graph) show that psmdev is the best performer up to 64-KByte messages, from then MVAPICH2 performs slightly better up to 256 KBytes, but then Open MPI becomes again the best large message performer. Regarding the MN supercomputer (bottom left graph), mxdev results are worse than the MPICH-MX ones up to 256 KBytes, but it shows quite competitive performance and scalability from this point on. Finally, on the AMD-SHM testbed (bottom right graph), smdev generally outperforms MVAPICH2 from 2-KByte messages and shows results quite close to Open MPI up to 256 KBytes, although Open MPI benefits, once again, from its better large message performance.

The presented results show that FastMPJ is generally able to obtain performance results for the broadcast operation similar to MPI libraries, even outperforming them in some message ranges. This supports the fact that the MST-based algorithm implemented in the FastMPJ collective library is very efficient (e.g., clearly outperforms MVAPICH on the AMD-SHM testbed) and highly scalable (e.g., large message performance using 1024 cores on the MN supercomputer). Therefore, these results confirm that FastMPJ is bridging the gap between MPJ and MPI collectives performance. Nevertheless, there is still potential room for improvement, especially for large message bandwidth, which means that enhanced collective algorithms and techniques need to be explored in order to achieve the high performance shown by Open MPI.

6.4 HPC Kernel Performance Analysis

The performance analysis of representative HPC kernels has been carried out using both strong (Section 6.4.1) and weak (Section 6.4.2) scaling models. The metrics considered for this evaluation using the NPB suite are MOPS (Millions of Operations Per Second), which measures the operations performed in the benchmark (and which differs from the CPU operations issued), and their corresponding speedups and efficiencies for the strong and weak scaling models, respectively.

6.4.1 Strong Scaling

In this first set of 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. These experiments have been conducted on the IB-QDR and IB-DDR testbeds, selected as they are the most representative distributed memory systems under evaluation. Thus, both multi-core clusters provide an IB interconnection network from the major current vendors (Mellanox and Intel/QLogic, respectively). Furthermore, in recent years, IB has become the most widely adopted networking technology in the TOP500 list. Additionally, both shared memory testbeds (Intel-SHM and AMD-SHM) have also been included in this analysis, as they provide with representative Intel- and AMD-based processors, respectively.

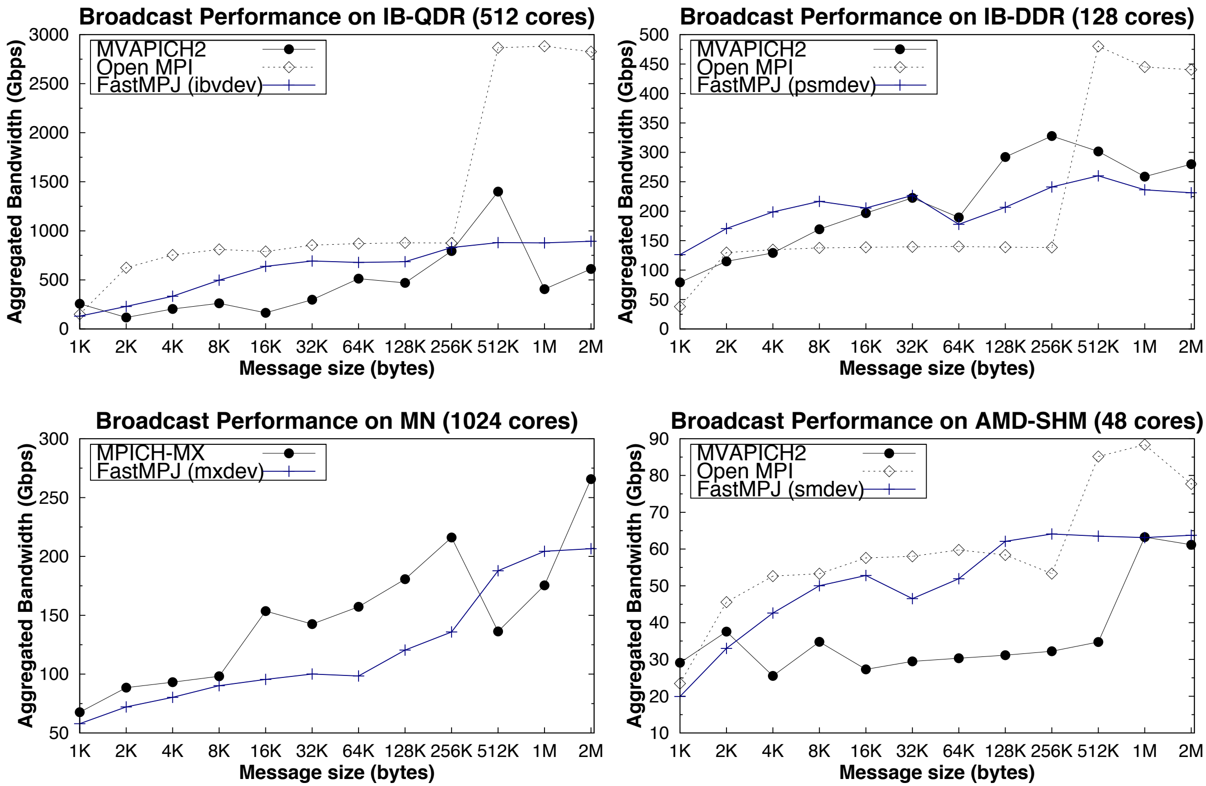

Fig. 4 Broadcast performance on InfiniBand QDR and DDR, Myrinet and shared memory

Figure 5 shows the NPB kernels performance on the IB-QDR testbed in terms of MOPS (left graphs) and its corresponding speedups (right graphs) using up to 512 cores. Regarding CG, FastMPJ and MPI show very similar results using up to 64 cores, as the scalability of this kernel is strongly based on point-to-point data transfers where FastMPJ and MPI achieve comparable performance, as has been observed before in the point-to-point micro-benchmarking. From 64 cores, `ibvdev` starts to suffer the current limitation of not being able to take advantage of intra-node communications, which seems to aggravate when the number of cores increases, as more communications have to be performed accessing the NIC instead of using a shared memory approach. This fact allows MPI libraries to obtain the highest performance and speedup from 128 cores on, but FastMPJ results remain competitive at least compared to MVAPICH2. FT results show that, while FastMPJ performance is on average 25% lower than MPI, the reported speedups are quite similar. In this case, FastMPJ performance is limited by its poor performance on a single core, as this kernel presents the largest performance gap between Java and native implementations (approximately 35% less performance). In addition, the FT kernel makes an intensive use of Alltoall collective operations, which has not prevented FastMPJ scalability. The performance and scalability of FastMPJ for the IS kernel is quite similar to Open MPI, although the maximum observed speedups are significantly low (below 60 on 256 cores for MVAPICH2). The implementation of this kernel relies heavily on Alltoall and Allreduce primitives, whose overhead is the main performance penalty, especially when using more than 256 cores on this testbed (all evaluated middleware drops in performance from this point). Finally, the MG kernel is the least communication-intensive code under evaluation; it shows relatively high speedups (above 300 on 512 cores) both for FastMPJ and MPI.

Figure 6 shows the NPB kernels performance on the IB-DDR testbed using up to 128 cores. CG results on this system show that FastMPJ is able to match the performance and speedup of MVAPICH2. In this scenario, all the middleware relies on the same underlying low-level communication sub-system (the PSM library). Thus, PSM implements the communication protocols and ultimately determines the point-to-point performance both for inter-node and intra-node communications, which prevents MPI libraries to use their own shared memory protocol (PSM already provides efficient shared memory support). Regarding the FT ker-

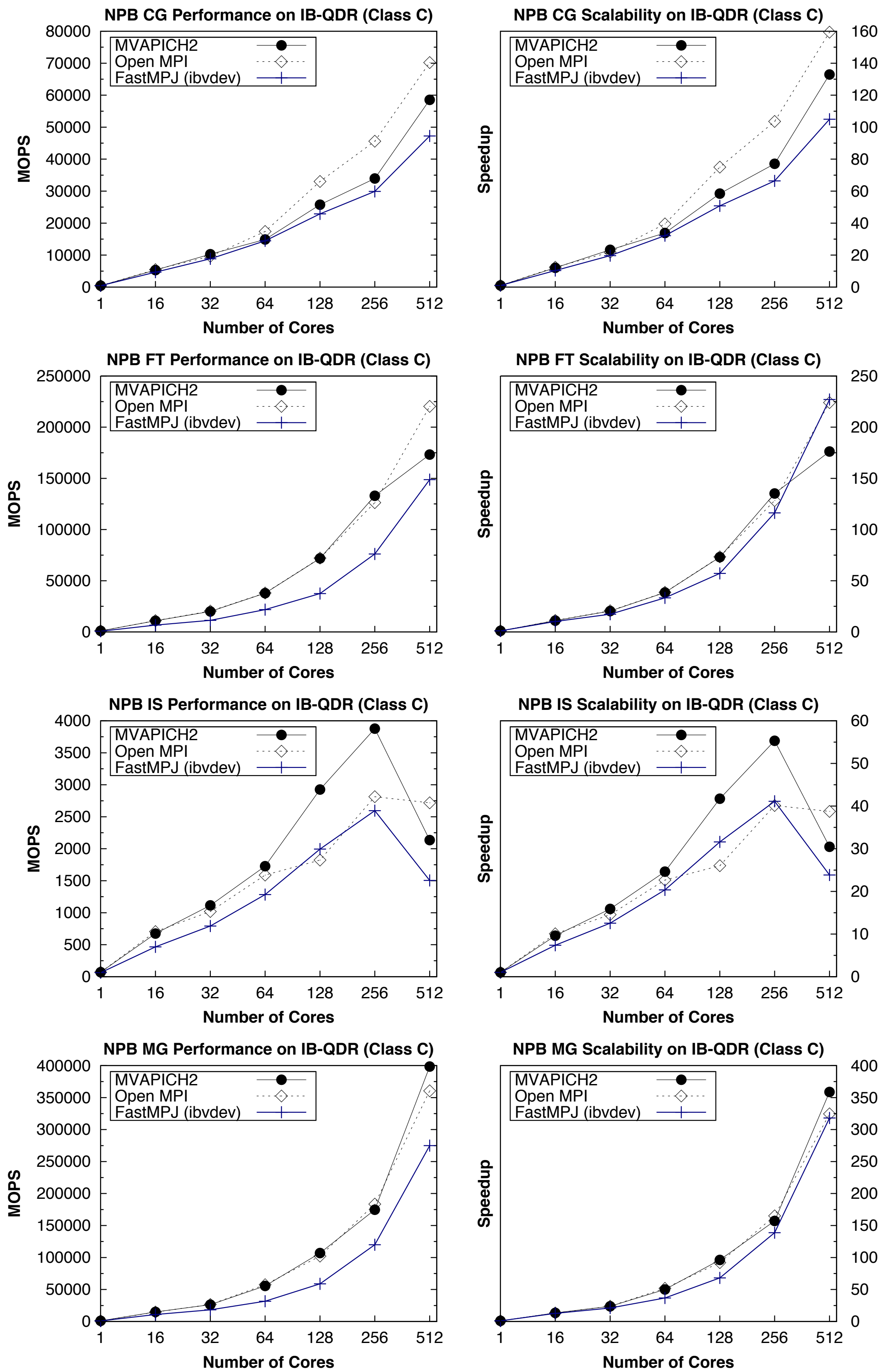

Fig. 5 NPB kernel results on the IB-QDR testbed (strong scaling)

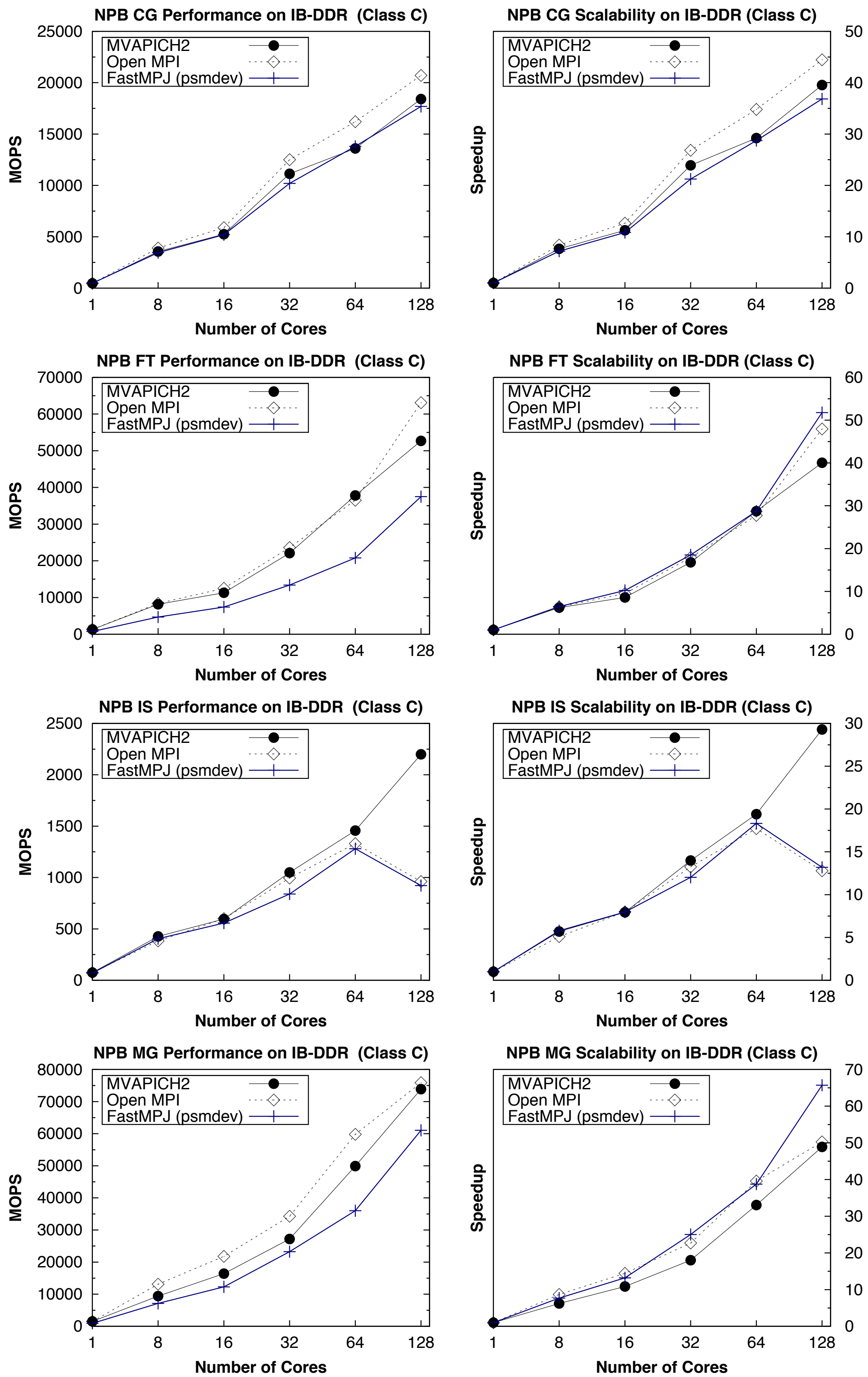

Fig. 6 NPB kernel results on the IB-DDR testbed (strong scaling)

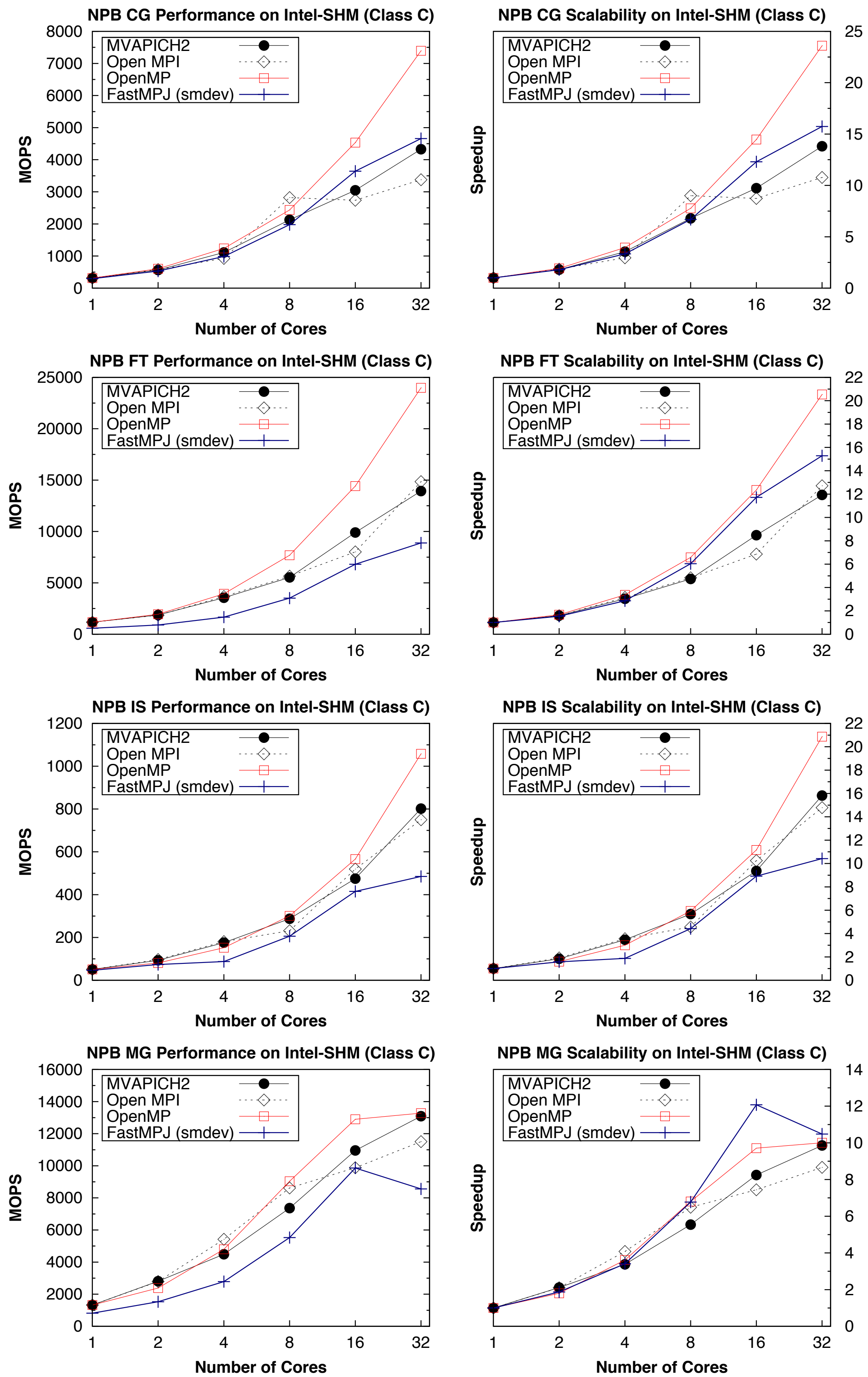

Fig. 7 NPB kernel results on the Intel-SHM testbed (strong scaling)

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

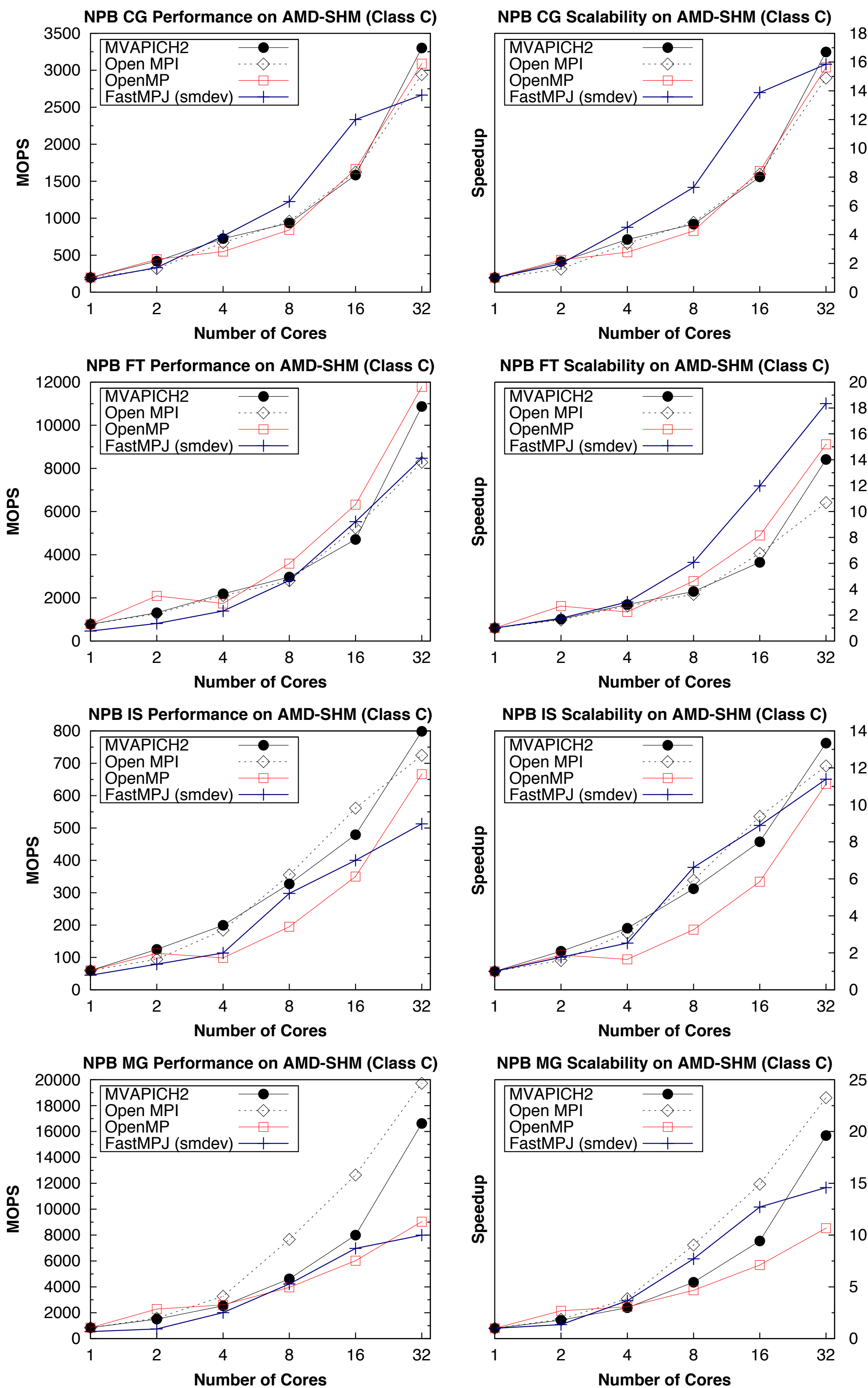

Fig. 8 NPB kernel results on the AMD-SHM testbed (strong scaling)

nel, FastMPJ obtains the highest speedup when using 128 cores, although its performance is around 30% lower than MVAPICH2 due to the poor Java serial performance, as mentioned before. The IS kernel shows again the poorest scalability (below 30 on 128 cores), where FastMPJ is able to achieve the same performance as MPI libraries using up to 64 cores. For MG, FastMPJ shows again the highest speedups, especially on 128 cores, motivated by the different serial runtime of the native and Java implementation (30% gap in this testbed). This also causes that FastMPJ obtains lower performance than MPI on 128 cores (around 20%).

Regarding shared memory systems, Figures 7 and 8 show the NPB kernels performance on the Intel-SHM and AMD-SHM testbeds, respectively, using up to 32 cores. The comparison on this scenario also includes the results from the OpenMP implementation of the NPB kernels. On the one hand, Intel-SHM results show that OpenMP is generally the best performer, both in terms of MOPS and scalability, except for the MG kernel where FastMPJ obtains the highest speedup. In addition, FastMPJ is able to achieve better performance than MPI for the CG kernel, taking advantage of the higher bandwidth obtained by `smdev`, whereas for the remaining kernels FastMPJ shows competitive results compared to MPI using up to 16 cores. On the other hand, results on the AMD-SHM testbed show that: (1) FastMPJ is able to outperform all the middleware for the CG kernel using up to 16 cores; (2) it obtains similar results as Open MPI for FT; and (3) it outperforms OpenMP and gets comparable performance to MVAPICH2 for the IS and MG kernels, using up to 16 cores. However, the AMD system generally obtains lower performance than the Intel system for all the evaluated middleware, due to its lower computational power per core and poorer memory access throughput, which limits the obtained speedups.

6.4.2 Weak Scaling

In the case of weak scaling, the problem size increases with the number of cores so that the workload per core remains constant. In our experiments, the NPB Class C are solved using a quarter of the number of available cores. Maintaining a fixed workload per core, results are reported from a workload of Class C divided by 8 up to 4 times Class C. Thus, the problem size is scaled linearly with the core count, as will be shown in the X-axis of the graphs (see Figures 9 and 10). This set of experiments has been conducted on the IB-QDR and Intel-SHM testbeds, selected as representative distributed and shared memory systems, respectively, which, according to the previous strong scaling evaluation, have shown the best performance results. In addition, as the NPB weak scaling results were, in general, quite similar to the previous strong scaling counterparts, both in terms of MOPS and speedups, only results for CG and FT kernels are shown for clarity purposes.

NPB weak scaling results are shown in MOPS (as in the case of strong scaling) together with their corresponding scaling efficiencies, instead of speedups. Note that the scaling efficiency metric has not been calculated as a percentage of the linear speedup, because usually can not be achieved. Instead, an upper bound on performance has been estimated for each core count using the serial code with the corresponding problem size. Thus, running multiple serial processes concurrently (as many processes per node as the number of cores under evaluation) takes into account the overhead associated with several processes accessing some shared levels of cache and memory bandwidth, which prevents obtaining the linear speedup. As an example, the upper bound performance for the FT kernel has achieved a speedup of 458 on 512 cores on IB-QDR, and 21 on 32 cores on Intel-SHM. Additionally, as there is no inter-process communication involved in the estimation of this value, it also represents an upper bound on performance if it were possible to perform zero-latency communications. Therefore, the efficiency of the corresponding parallel code calculated as a percentage of this estimated upper bound value can serve as a reliable metric to measure the communication efficiency of message-passing libraries. As there are no explicit communication routines in the OpenMP standard, NPB-OMP results are not shown in the Intel-SHM testbed.

NPB results on the IB-QDR and Intel-SHM testbeds are presented in Figures 9 and 10, respectively. On the one hand, the CG kernel shows that Java can obtain an upper bound performance quite similar to Fortran when no communication is involved, especially in the Intel-SHM testbed. In this scenario, FastMPJ is able to almost match the performance of at least one of the MPI libraries on both testbeds (MVAPICH2 on IB-QDR and OpenMPI on Intel-SHM). Consequently, the communication efficiency of FastMPJ is in tune with MPI libraries, as shown in the right graphs, especially for the higher core counts. On the other hand, the FT kernel results show that in this case the upper bound performance for Java is limited by its poor performance on a single core, which is on average around 60% of Fortran's performance. The main performance penalty is the lack of a high-performance numerical library in Java that would implement the Fourier transform, which is the most computationally intensive part of this kernel. However, while FastMPJ performance is on average 20% lower than MPI for the higher core

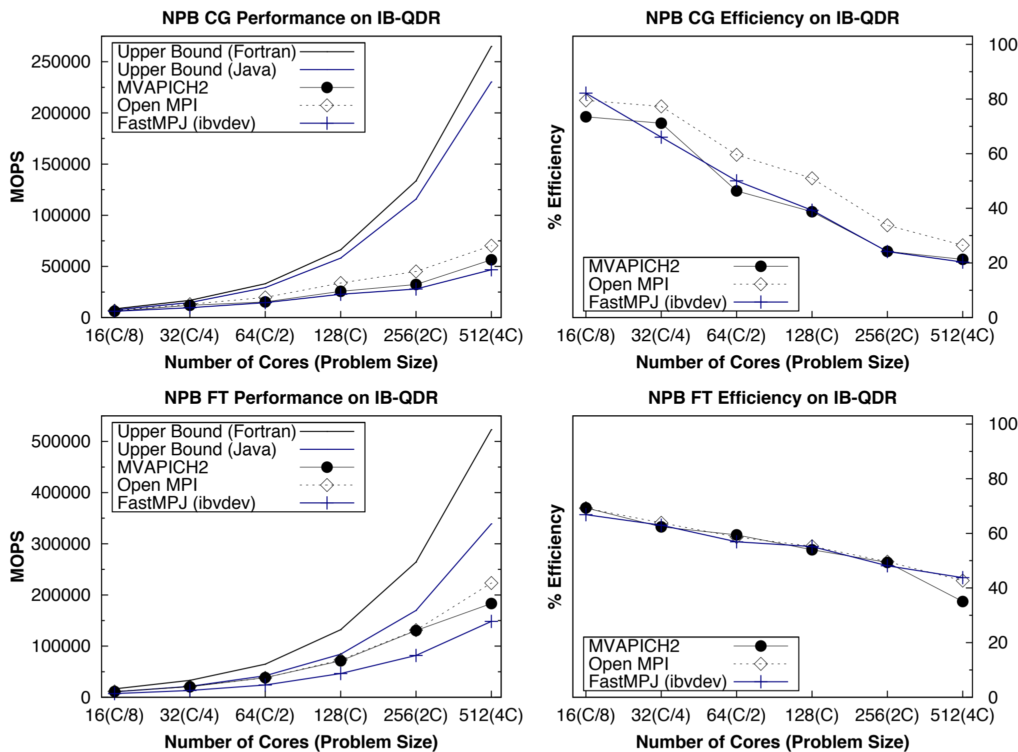

Fig. 9 NPB kernel results on the IB-QDR testbed (weak scaling)

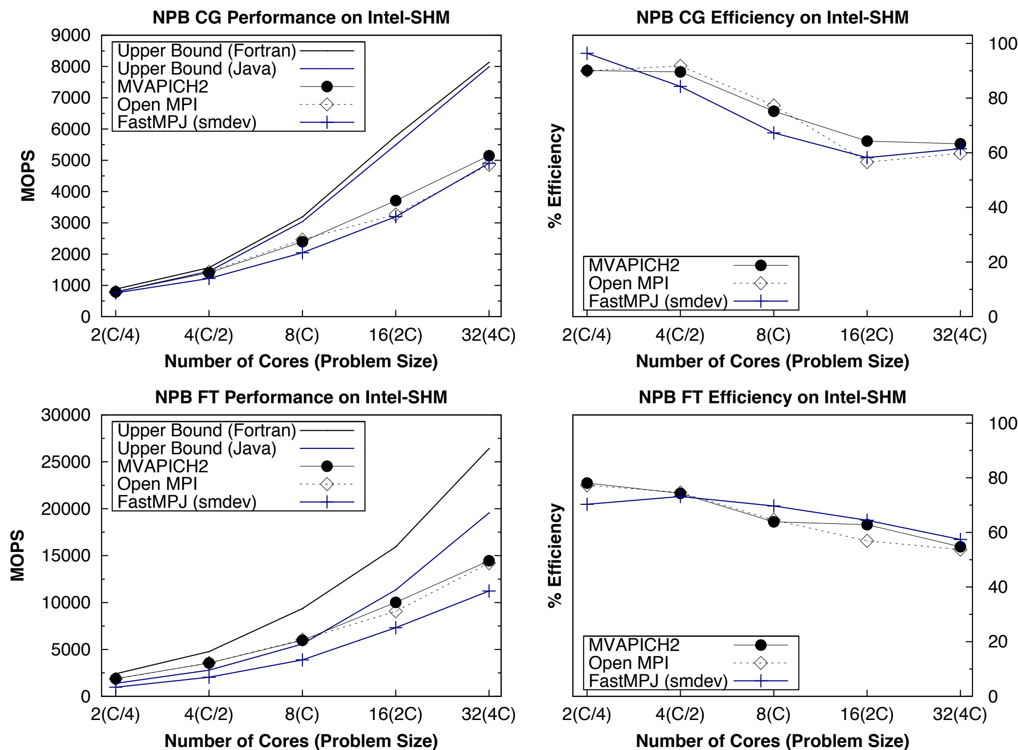

Fig. 10 NPB kernel results on the Intel-SHM testbed (weak scaling)

counts, the reported efficiencies are quite similar. Thus, this fact confirms that the underlying communication support implemented by FastMPJ is able to achieve comparable performance to MPI.

To sum up, the NPB results using both scaling metrics have shown that FastMPJ is able to rival native MPI performance and scalability, even outperforming MPI in some scenarios (e.g., CG kernel on IB-DDR and shared memory systems). This allows Java to take advantage of the use of a high number of cores, especially on shared memory and hybrid shared/distributed memory architectures, widely extended nowadays.

7 Conclusions

The continuous increase in the number of cores per system underscores the need for scalable parallel solutions both in shared and distributed memory architectures, where the efficiency of the underlying communication middleware is fundamental. In fact, the scalability of Java message-passing parallel applications depends heavily on the communications performance. However, current Java communication middleware lacks efficient communication support, especially in the presence of high-speed cluster networks and shared memory systems.

This paper has presented FastMPJ, a scalable and efficient Java message-passing library for parallel computing, which overcomes these performance constraints by: (1) providing thread-based high-performance shared memory communications which obtains sub-microsecond start-up latencies and up to 71.2 Gbps bandwidth; (2) enabling low-latency (less than 2 μs) and high bandwidth communications (higher than 22 Gbps) on RDMA-capable high-speed cluster networks (e.g., InfiniBand); (3) including a scalable collective library with more than 60 topology aware algorithms, which are automatically selected at runtime; (4) avoiding Java data buffering overheads through efficient zero-copy protocols; and (5) implementing the mpiJava 1.2 API, the most widely extended MPI-like Java bindings, for a highly productive development of MPJ parallel applications.

FastMPJ has been evaluated comparatively with native MPI libraries on five representative testbeds: two InfiniBand multi-core clusters, one Myrinet supercomputer, and two shared memory systems using both Intel- and AMD-based processors. The comprehensive performance evaluation has revealed that FastMPJ communication primitives are quite competitive with MPI results, both in terms of point-to-point and collective operations performance. Thus, the use of our message-passing library in communication-intensive HPC codes allows Java to benefit from a more efficient communication support, taking advantage of the use of a high number of cores and improving significantly the performance and scalability of Java parallel applications. In fact, the development of this efficient Java communication middleware is definitely bridging the gap between Java and native languages in HPC applications. Further information of this project is available at http://torusware.com.

Acknowledgements This work has been funded by the Ministry of Education of Spain (FPU grant AP2010-4348), the Ministry of Economy and Competitiveness (project TIN2010-16735) and the Galician Government (projects CN2012/211 and GRC2013/055), partially supported by FEDER funds. We thankfully acknowledge the computer resources, technical expertise and assistance provided by the Barcelona Supercomputing Center. We also gratefully thank the Advanced School for Computing and Imaging (ASCI) and the Vrije University Amsterdam for providing access to the DAS-4 cluster.

References

1. Java Grande Forum. http://www.javagrande.org. Accessed: October 2013
2. MareNostrum supercomputer in TOP500 List. http://www.top500.org/system/8242. Accessed: October 2013
3. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/. Accessed: October 2013
4. NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html. Accessed: October 2013
5. Open MPI: Open Source High Performance Computing. http://www.open-mpi.org/. Accessed: October 2013
6. Portable MPI Model Implementation over MX. https://www.myricom.com/support/downloads/mx/mpich-mx.html. Accessed: October 2013
7. Advanced School for Computing and Imaging (ASCI): Distributed ASCI Supercomputer - Version 4 (DAS-4). http://www.cs.vu.nl/das4/. Accessed: October 2013
8. Bailey, D. H., et al.: The NAS parallel benchmarks. International Journal of High Performance Computing Applications **5**(3), 63–73 (1991)
9. Baker, M., Carpenter, B., Fox, G., Ko, S.H., Lim, S.: mpiJava: An object-oriented Java interface to MPI. In: Proc. of 1st International Workshop on Java for Parallel and Distributed Computing (IWJPDC'99), pp. 748–762. San Juan, Puerto Rico (1999)
10. Baker, M., Carpenter, B., Shafi, A.: A pluggable architecture for high-performance Java messaging. IEEE Distributed Systems Online **6**(10) (2005)
11. Baker, M., Carpenter, B., Shafi, A.: MPJ Express: Towards thread safe Java HPC. In: Proc. of 8th IEEE International Conference on Cluster Computing (CLUSTER'06), pp. 1–10. Barcelona, Spain (2006)
12. Baker, M., Carpenter, B., Shafi, A.: A buffering layer to support derived types and proprietary networks for Java HPC. Scalable Computing Practice and Experience **8**(4), 343–358 (2007)
13. Blount, B., Chatterjee, S.: An evaluation of Java for numerical computing. Scientific Programming **7**(2), 97–110 (1999)

14. Bonachea, D., Dickens, P.M., Thakur, R.: High-performance file I/O in Java: Existing approaches and bulk I/O extensions. Concurrency and Computation: Practice and Experience **13**(8-9), 713–736 (2001)
15. Bornemann, M., van Nieuwpoort, R.V., Kielmann, T.: MPJ/Ibis: A flexible and efficient message passing platform for Java. In: Proc. of 12th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'05), pp. 217–224. Sorrento, Italy (2005)
16. Carpenter, B., Fox, G., Ko, S.H., Lim, S.: mpiJava 1.2: API specification. http://www.hpjava.org/reports/mpiJava-spec/mpiJava-spec/mpiJava-spec.html. Accessed: October 2013
17. Carpenter, B., Getov, V., Judd, G., Skjellum, A., Fox, G.: MPJ: MPI-like message passing for Java. Concurrency and Computation: Practice and Experience **12**(11), 1019–1038 (2000)
18. Chan, E., Heimlich, M., Purkayastha, A., van de Geijn, R.A.: Collective communication: Theory, practice, and experience. Concurrency and Computation: Practice and Experience **19**(13), 1749–1783 (2007)
19. Dickens, P.M., Thakur, R.: An evaluation of Java's I/O capabilities for high-performance computing. In: Proc. of 1st ACM Java Grande Conference (JAVA'00), pp. 26–35. San Francisco, CA, USA (2000)
20. Expósito, R.R., Taboada, G.L., Touriño, J., Doallo, R.: Design of scalable Java message-passing communications over InfiniBand. Journal of Supercomputing **61**(1), 141–165 (2012)
21. Goglin, B.: High throughput intra-node MPI communication with Open-MX. In: Proc. of 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP'09), pp. 173–180. Weimar, Germany (2009)
22. Goglin, B.: High-performance message passing over generic Ethernet hardware with Open-MX. Parallel Computing **37**(2), 85–100 (2011)
23. Hongwei, Z., Wan, H., Jizhong, H., Jin, H., Lisheng, Z.: A performance study of Java communication stacks over InfiniBand and Gigabit Ethernet. In: Proc. 4th IFIP International Conference on Network and Parallel Computing - Workshops (NPC'07), pp. 602–607. Dalian, China (2007)
24. IBTA: The InfiniBand Trade Association. http://www.infinibandta.org/. Accessed: October 2013
25. IETF RFC 4392: IP over InfiniBand (IPoIB) Architecture. http://www.ietf.org/rfc/rfc4392.txt.pdf. Accessed: October 2013
26. Mallón, D.A., Taboada, G.L., Touriño, J., Doallo, R.: NPB-MPJ: NAS parallel benchmarks implementation for message-passing in Java. In: Proc. of 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP'09), pp. 181–190. Weimar, Germany (2009)
27. Message Passing Interface Forum: MPI: A Message Passing Interface standard. http://www.mcs.anl.gov/research/projects/mpi/ (1995). Accessed: October 2013
28. Myrinet Express (MX): A high performance, low-level, message-passing interface for Myrinet. Version 1.2, October 2006
29. van Nieuwpoort, R.V., Maassen, J., Wrzesinska, G., Hofman, R., Jacobs, C., Kielmann, T., Bal, H.E.: Ibis: A flexible and efficient Java-based Grid programming environment. Concurrency and Computation: Practice and Experience **17**(7-8), 1079–1107 (2005)
30. OpenFabrics Alliance: http://www.openfabrics.org/. Accessed: October 2013
31. Ramos, S., Taboada, G.L., Expósito, R.R., Touriño, J., Doallo, R.: Design of scalable Java communication middleware for multi-core systems. The Computer Journal **56**(2), 214–228 (2013)
32. Saini, S. et al.: Performance evaluation of supercomputers using HPCC and IMB benchmarks. Journal of Computer and System Sciences **74**(6), 965–982 (2008)
33. 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 **21**(15), 1882–1906 (2009)
34. Shafi, A., Manzoor, J., Hameed, K., Carpenter, B., Baker, M.: Multicore-enabling the MPJ Express messaging library. In: Proc. of 8th International Conference on the Principles and Practice of Programming in Java (PPPJ'10), pp. 49–58. Vienna, Austria (2010)
35. 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 **39**(1), 175–193 (2000)
36. Taboada, G.L., Ramos, S., Expósito, R.R., Touriño, J., Doallo, R.: Java in the high performance computing arena: Research, practice and experience. Science of Computer Programming **78**(5), 425–444 (2013)
37. Taboada, G.L., Touriño, J., Doallo, R.: Java Fast Sockets: Enabling high-speed Java communications on high performance clusters. Computer Communications **31**(17), 4049–4059 (2008)
38. Taboada, G.L., Touriño, J., Doallo, R.: F-MPJ: Scalable Java message-passing communications on parallel systems. Journal of Supercomputing **60**(1), 117–140 (2012)
39. Taboada, G.L., 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 **23**(18), 2382–2403 (2011)
40. Thiruvathukal, G.K., Dickens, P.M., Bhatti, S.: Java on networks of workstations (JavaNOW): A parallel computing framework inspired by Linda and the Message Passing Interface (MPI). Concurrency and Computation: Practice and Experience **12**(11), 1093–1116 (2000)
41. TOP500 Org.: Top 500 Supercomputer Sites. http://www.top500.org/. Accessed: October 2013

Roberto R. Expósito received the B.S. (2010) and M.S. (2011) degrees in Computer Science from the University of A Coruña, Spain. Currently he is a Ph.D. student in the Department of Electronics and Systems at the University of A Coruña. His research interests are in the area of High Performance Computing (HPC), focused on message-passing communications on high-speed cluster networks and cluster/cloud computing.

Sabela Ramos received the B.S. (2009), M.S. (2010) and Ph.D. (2013) degrees in Computer Science from the University of A Coruña, Spain. Currently she is a Teaching Assistant in the Department of Electronics and Systems at the University of A Coruña. Her research interests are in the area of High Performance Computing (HPC), focused on message-passing communications on multicore architectures and cluster/cloud computing. Her homepage is http://gac.udc.es/~sramos.

Guillermo L. Taboada received the B.S. (2002), M.S. (2004) and Ph.D. (2009) degrees in Computer Science from the University of A Coruña, Spain. Currently he is an Associate Professor in the Department of Electronics and Systems at the University of A Coruña. His main research interest is in the area of High Performance Computing (HPC), focused on high-speed networks, programming languages for HPC, cluster/cloud computing and, in general, middleware for HPC. He is coauthor of more than 25 technical papers on these topics. His homepage is http://gac.udc.es/~gltaboada.

Juan Touriño received the B.S. (1993), M.S. (1993) and Ph.D. (1998) degrees in Computer Science from the University of A Coruña, Spain. In 1993 he joined the Department of Electronics and Systems at the University of A Coruña, where he is currently a Full Professor of computer engineering. He has extensively published in the areas of compilers and programming languages for High Performance Computing (HPC), and parallel and distributed computing. He is coauthor of more than 120 technical papers on these topics. His homepage is http://gac.udc.es/~juan.

Ramón Doallo received the B.S. (1987), M.S. (1987) and Ph.D. (1992) degrees in Physics from the University of Santiago de Compostela, Spain. In 1990 he joined the Department of Electronics and Systems at the University of A Coruña, Spain, where he became a Full Professor in 1999. He has extensively published in the areas of computer architecture, and parallel and distributed computing. He is coauthor of more than 140 technical papers on these topics. His homepage is http://gac.udc.es/~doallo.