CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
*Concurrency Computat.: Pract. Exper.* (2011)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1777

# Device level communication libraries for high-performance computing in Java

Guillermo L. Taboada [1,*,†], Juan Touriño [1], Ramón Doallo [1], Aamir Shafi [2], Mark Baker [3] and Bryan Carpenter [4]

[1] *Computer Architecture Group, University of A Coruña, A Coruña, Spain*
[2] *MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA*
[3] *School of Systems Engineering, University of Reading, Reading, UK*
[4] *School of Computing, University of Portsmouth, Portsmouth, UK*

## SUMMARY

Since its release, the Java programming language has attracted considerable attention from the high-performance computing (HPC) community because of its portability, high programming productivity, and built-in multithreading and networking support. As a consequence, several initiatives have been taken to develop a high-performance Java message-passing library to program distributed memory architectures, such as clusters. The performance of Java message-passing applications relies heavily on the communications performance. Thus, the design and implementation of low-level communication devices that support message-passing libraries is an important research issue in Java for HPC. MPJ Express is our Java message-passing implementation for developing high-performance parallel Java applications. Its public release currently contains three communication devices: the first one is built using the Java New Input/Output (NIO) package for the TCP/IP; the second one is specifically designed for the Myrinet Express library on Myrinet; and the third one supports thread-based shared memory communications. Although these devices have been successfully deployed in many production environments, previous performance evaluations of MPJ Express suggest that the buffering layer, tightly coupled with these devices, incurs a certain degree of copying overhead, which represents one of the main performance penalties. This paper presents a more efficient Java message-passing communications device, based on Java Input/Output sockets, that avoids this buffering overhead. Moreover, this device implements several strategies, both in the communication protocol and in the HPC hardware support, which optimizes Java message-passing communications. In order to evaluate its benefits, this paper analyzes the performance of this device comparatively with other Java and native message-passing libraries on various high-speed networks, such as Gigabit Ethernet, Scalable Coherent Interface, Myrinet, and InfiniBand, as well as on a shared memory multicore scenario. The reported communication overhead reduction encourages the upcoming incorporation of this device in MPJ Express (http://mpj-express.org). Copyright © 2011 John Wiley & Sons, Ltd.

Received 30 July 2010; Revised 11 April 2011; Accepted 12 April 2011

KEY WORDS: Message Passing in Java (MPJ); Java for high-performance computing (HPC); high-speed networks; shared memory multicore communication; performance evaluation

## 1. INTRODUCTION

The Java programming language has now become a leading platform in the software industry as it allows developers to write portable, safe, robust, and reliable multithread and network-based applications. Moreover, there has been a continuous and growing interest in Java for high-performance computing (HPC) [1, 2]. This interest is based on several appealing characteristics of Java, which include its built-in networking and multithreading support, object orientation and thus higher

*Correspondence to: Guillermo L. Taboada, Computer Architecture Group, University of A Coruña, A Coruña, Spain.

†E-mail: taboada@udc.es

programming productivity, platform independence, portability, and security. These significant benefits motivated the appearance of the Java Grande Forum [3], an initiative devoted to promote the use of Java for Grande applications, those with large requirements of computational resources, and proposed modifications to the Java language specification to make it more suitable for these codes. Furthermore, in the era of multicore processors, the use of Java threads is considered to be a feasible option to harness the performance of these processors. Another interesting argument in favor of Java is the large pool of developers, especially due to its significant presence in academia.

Java, in its early days, was severely criticized for its poor computational performance [4], but the performance gap between Java and native (compiled) languages like C and C++ has narrowed significantly. The main reason is that the Java Virtual Machine (JVM), which executes Java applications, is now equipped with just-in-time compilers. The JVM identifies sections of the code frequently executed and converts them to native machine code instead of interpreting the bytecode. Nevertheless, the tremendous improvement in its computational performance is not enough for Java to be a successful language in the area of parallel computing, as its communications performance is also essential to obtain scalability in Java for HPC.

Traditionally the HPC community has adopted two mainstream architectures: shared and distributed memory. In the shared memory approach, all processors communicate to each other via shared memory transfers, whereas for distributed memory architectures, processors must resort to message passing for sharing data. Nevertheless, the emergence of multicore processors has had a significant impact on HPC hardware platforms. Currently, the most popular architectures are high-speed clusters, several compute nodes interconnected to each other via a high-speed network, thanks to their interesting cost/performance ratio. In fact, current cluster deployments are increasingly relying on compute nodes with a high number of cores in order to meet the ever growing computational power needs. Such hybrid hardware underscores the importance of parallelism and multithreading [5], where only a hybrid shared/distributed memory programming model can fully exploit the available processing power. Here, Java represents a practical and attractive choice for programming these systems as it supports both message passing and built-in multithreading. As Java has built-in multithreading support, the sustained interest in the use of Java for HPC has motivated the development of an important number of message-passing libraries [2].

In hybrid shared/distributed memory architectures, the approach that is considered to take full advantage of the underlying hardware is the simultaneous use of the message-passing paradigm together with thread-based solutions. However, although this hybrid programming approach can provide good performance, it has, as a main drawback, the fact that it requires the use of two programming paradigms, increasing the complexity of parallel programming. Thus, the productivity in the development of parallel applications motivates that a significant number of applications are only implemented using the message-passing model. Nevertheless, the efficiency of this pure message-passing approach (without multithreading) to program multicore clusters depends on its intra-node communications support. Thus, current Message Passing in Java (MPJ) libraries are developing their shared memory communication support, based on either sockets, native inter-process communication libraries, or Java threads.

MPJ Express [6] is an implementation of the Java bindings for the Message-Passing Interface (MPI) standard [7]. This message-passing library also provides thread-safe communication in order to support hybrid shared/distributed memory programming. Moreover, it targets both performance and portability by supporting pluggable communication devices. Thus, MPJ Express includes a fully portable ‘pure’ (100%) Java device based on Java New Input/Output (NIO) sockets on TCP/IP, together with various communication devices targeted to high-performance hardware (Myrinet network and multicore nodes).

There is a growing community of users who are developing their parallel applications using MPJ Express. Recently, researchers at National Aeronautics and Space Administration Langley used the software to parallelize their 3D radiative transfer modeling application [8]. Another scientific software using MPJ Express is CartaBlanca [9], an object-oriented physical system simulation package. The code uses Jacobian-free Newton–Krylov methods to solve nonlinear physics simulations on unstructured meshes. Additionally, Gadget, a very popular code for cosmological $N$-body/smoothed-particle hydrodynamics simulations, and ProtTest, a widely extended

DOI: 10.1002/cpe

bioinformatics application for the selection of best-fit models of protein evolution, have also been implemented using MPJ Express within their Gadget [10] and ProtTest 3 [11] projects, respectively. Furthermore, the Modelling and Simulation in e-Social Science (MoSeS) project [12] at the University of Leeds is using MPJ Express.

This paper focuses on studying the communication performance of MPJ Express, the most representative Java message-passing implementation, on popular HPC interconnects and shared memory multicore systems. During this evaluation, and from our previous work [6], several potential bottlenecks have been identified, especially the use of a buffering layer that incurs a significant performance penalty. Thus, a new communication device without buffering overhead has been implemented. Moreover, this device implements several strategies, both in the communication protocol and the HPC hardware support, which definitely optimizes Java message-passing communications. This solution is thread safe; thus, shared memory and network communication solutions can be efficiently combined, which is of special interest for developing efficient parallel programs for the current mainstream multicore-based architecture and clusters with high-speed networks. The success of this device supports its upcoming incorporation in MPJ Express (http://mpj-express.org).

The rest of the paper is organized as follows. Section 2 presents the design of the communications support in MPJ Express with a special emphasis on its low-level communication device and buffering layers. This is followed by Section 3, which presents the new communication device meant for reducing the message-passing communication overhead in MPJ Express. Section 4 evaluates the performance of the new device on shared memory and various HPC interconnects. Section 5 analyzes its impact on the overall performance of MPJ applications. Section 6 comments on related work. Finally, Section 7 concludes the paper.

## 2. MPJ EXPRESS COMMUNICATION DEVICES DESIGN

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

Figure 1 illustrates the MPJ Express design and the different levels of the software. The topmost MPJ API layer presents the full API of the library. The next two layers contain the high level and base level primitives, representing the collective and point-to-point communications, respectively. The point-to-point primitives (or base level) are implemented on top of `mpjdev`, the MPJ device layer [13], which has two implementations, the ‘pure’ (100%) Java and the native one. The pure Java `mpjdev` relies on the `xdev` low-level communications layer for actual communications and interaction with the underlying networking hardware, whereas the native `mpjdev` is a research library on top of a native MPI implementation.

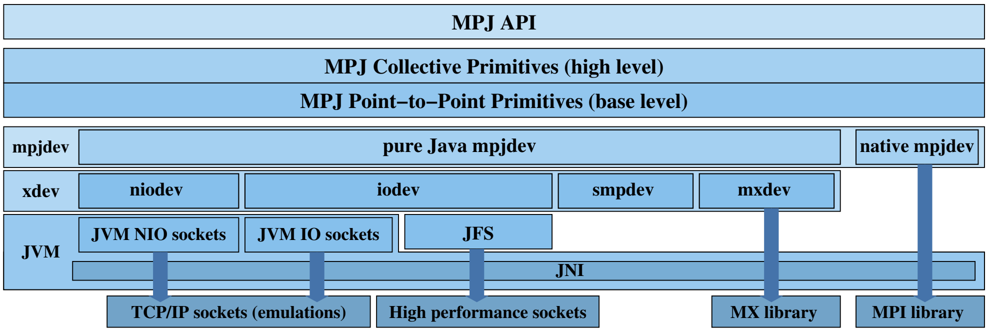

Figure 1. Overview of the MPJ Express design. MPJ, Message Passing in Java; JFS, Java Fast Sockets; IO, Input/Output; NIO, New IO; JNI, Java Native Interface; MX, Myrinet Express; MPI, Message-Passing Interface.

DOI: 10.1002/cpe

The main reason behind the introduction of a second device layer, xdev, in the MPJ Express design is to reduce the effort when implementing the support for new underlying communication libraries. Unlike the mpjdev layer, xdev only provides basic point-to-point communication methods and is not aware of higher-level MPI abstractions like communicators. This pluggable design, the combination of xdev and mpjdev, provides higher-level libraries with an already known API (mpjdev), while adapting through custom implementations (xdev) to specific HPC interconnection hardware, easing the integration, as well as the use and efficiency of the solution [14].

In order to favor the development of the new xdev communication devices, it has been defined an abstract class xdev.Device, which provides an xdev API (see Subsection 2.1) to which all device implementations must conform. This pluggable design allows for runtime selection of the most appropriate communication device.

There are four specialized communication devices in the xdev layer. Three of them, niodev, smpdev, and mxdev, are currently bundled with the MPJ Express distribution and deployed in production systems, whereas the actual incorporation of iodev is the focus of this paper. On the one hand, niodev is a Java NIO-based device that relies on SocketChannel objects to implement point-to-point primitives, whereas smpdev [15] is a thread-based shared memory device that performs communications as intra-process transfers. On the other hand, mxdev uses Java Native Interface (JNI) to call MX native methods, a collection of specialized routines that are implemented to take full advantage of Myrinet-based hardware. Finally, iodev runs on top of Java IO socket implementations, both JVM libraries and high-performance ones, such as our Java Fast Sockets (JFS) [16], a Java TCP IO socket implementation that can access directly high-performance native socket libraries and thus take advantage of RDMA-capable interconnects, such as Scalable Coherent Interface (SCI), Myrinet, and InfiniBand, which are able to transfer efficiently bulk memory between nodes, while freeing the host CPU from communications processing.

These four xdev devices access HPC hardware through a variety of communication libraries that can be 'pure' (100%) Java, such as Java NIO and IO sockets, or that can be native libraries, such as high-performance sockets and MX. Thus, these devices can be classified into 'pure' Java devices, when they resort to JVM libraries (or other Java libraries), and wrapper devices, whose implementation is directly based on native methods, accessed through JNI. A 'pure' Java device is fully portable, whereas a device that relies on native libraries depends on the presence of those libraries in the target machine. The new iodev device combines both approaches as it accesses high-performance native sockets through JFS, when available, while it is portable, as it can always resort to JVM IO sockets.

Figure 2 presents the communications support of the two socket-based xdev implementations, niodev and iodev, on shared memory and several high-speed network interface cards (NIC): Gigabit Ethernet, SCI, Myrinet, and InfiniBand. This graph focuses on the different high-performance native libraries supported by each socket library. Thus, on the one hand, JVM NIO and IO sockets are supported by TCP/IP sockets on shared memory and Gigabit Ethernet and IP emulations on SCI, Myrinet, and InfiniBand, namely SCIP, IPoMX, and IPoIB, respectively. Nevertheless, on the other hand, JFS provides support on UNIX sockets for shared memory transfers and high-performance native socket libraries on SCI, Myrinet, and InfiniBand, namely SCI Sockets, Sockets-MX, and Sockets Direct Protocol, respectively. With respect to the performance that can be achieved, IP emulations usually provide a wider support but incur a higher communication overhead than high-performance native sockets, which are currently available through the use of JFS. Thus, iodev, which relies on JFS, can obtain higher performance than niodev thanks to its more efficient support of the underlying communication libraries. This paper includes in Section 4 a performance evaluation of these four xdev communication devices on all these environments.

### *2.1.* xdev *API design*

The xdev API (see Listing 1) has been designed with the goal of being simple and small, providing only basic communication methods in order to ease the development of xdev devices. An xdev communication device is similar to the MPI communicator class but with reduced functionality. The

DOI: 10.1002/cpe

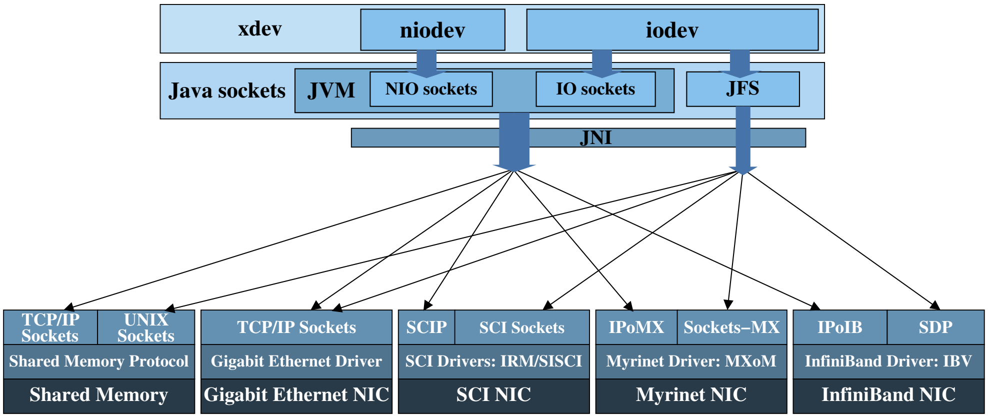

Figure 2. Overview of socket-based xdev devices support on HPC hardware. IO, Input/Output; NIO, New IO; JFS, Java Fast Sockets; JNI, Java Native Interface; SCI, Scalable Coherent Interface; IRM/SISCI, Interconnect Resource Manager/Software Infrastructure for SCI; NIC, Network Interface Cards; MX, Myrinet Express; IPoMX, IP over MX; MXoM, MX over Myrinet; IPoIB, IP over InfiniBand; SDP, Sockets Direct Protocol; IBV, InfiniBand Verbs.

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

  Request isend(mpjbuf.Buffer buf, ProcessID  dstID, int tag, int context);
  Request irecv(mpjbuf.Buffer buf, ProcessID  srcID, int tag, int context,
                Status status);
  void send(mpjbuf.Buffer buf, ProcessID  dstID, int tag, int context);
  Status recv(mpjbuf.Buffer buf, ProcessID  srcID, int tag, int context);

  Request issend(mpjbuf.Buffer buf, ProcessID  dstID, int tag, int context);
  void ssend(mpjbuf.Buffer buf, ProcessID srcID, int tag, int context);

  Status iprobe(ProcessID srcID, int tag, int context);
  Status probe(ProcessID srcID, int tag, int context);
  Request peek();

  int getSendOverhead();
  int getRecvOverhead();
}
```

Listing 1. API of the xdev.Device class.

init method starts the communication device operation. The id method returns the identification (ProcessID) of the device. The finish method is the last method to be called and completes the device operation.

The xdev communication primitives only include point-to-point communication, both blocking (send and recv, like MPI_Send and MPI_Recv) and nonblocking (isend and irecv like MPI_Isend and MPI_Irecv). Synchronous communications are also embraced (ssend and issend). MPJ Express implements the four communication modes (standard, synchronous, ready, and buffered) on top of xdev primitives. These communication methods use ProcessID objects instead of using ranks as arguments to send and receive primitives. In fact, the xdev layer is

DOI: 10.1002/cpe

focused on providing basic communication methods, and it does not deal with high-level message-passing abstractions such as groups, communicators, and contexts. Therefore, a `ProcessID` object unequivocally identifies a device object. It is the `mpjdev` layer (see Figure 1) that deals with communicators and groups management.

### 2.2. *MPJ Express buffering layer*

MPJ Express currently uses a buffering layer (implemented using Java NIO *direct* byte buffers) for three reasons:

1. `mpjdev` communication methods handle both primitive data types and object communications through the use of buffers, whose storage format has been already defined in objects of class `mpjbuf.Buffer`. The `xdev` layer also uses the same buffers for compatibility reasons.
2. The `niodev` device transfers messages through Java NIO `SocketChannel` objects that can only transfer byte buffers (`ByteBuffer` objects).
3. The native communication devices, such as `mxdev`, rely on the use of *direct* byte buffers as they reside inside the native operating system (OS) memory, avoiding JNI copying as well as supporting RDMA transfers on high-speed networks, thus reducing the communication overhead.

Although the specification does not define how a message is contained in a buffer, it states that the user is responsible for ensuring enough space to pack/unpack a message. Otherwise, an exception is thrown. `mpjbuf.Buffer` is divided into a primary payload, used to store primitive data type elements, and a secondary payload, for serialized objects. The size of the primary payload is static, whereas the secondary payload is dynamic, increasing its size with the number of objects that are written to the buffer. The primary payload is divided into sections. Each section contains elements of the same primitive data type. The secondary payload stores data according to the serialization specification. Although `xdev` and `mpjdev` share the storage format, `xdev` has implemented its own buffering strategy [17] to manage the communication buffers. The selected storage objects are Java NIO *direct* byte buffers, which allow performing of native IO operations directly upon them. However, because of their high allocation/deallocation times, a buffering scheme that consists of pooling reusable message buffers has been implemented. These message buffers are slices of the original buffers, thus allowing merges and splits of buffers in order to maximize their reusability. However, despite these optimizations, the use of this buffering layer incurs a copying overhead, that can be significant for large messages.

## 3. IMPROVING THE MPJ EXPRESS PERFORMANCE: *IODEV* DEVICE

Although MPJ Express has been successfully deployed in many production environments, our previous performance evaluations [17, 18] suggest that the buffering layer incurs a certain degree of copying overhead, representing one of its main performance penalties. This copying overhead is caused by the use of the `xdev` API, which is tightly coupled to the buffering layer. In fact, every message (which can be either an object or a primitive data type array) must be packed into an `mpjbuf.Buffer` object in order to be sent and subsequently be unpacked at the receiver side into the destination object. These pack/unpack (copying) operations form the major performance bottlenecks in MPJ Express, as experimentally assessed in Section 4, thus limiting significantly its scalability.

Therefore, in order to overcome this issue, we have implemented `iodev`, a low-level communication device whose API extends the `xdev` API, allowing the communication of any serializable object, not only `mpjbuf.Buffer` objects. Thus, the buffering of data in an `mpjbuf.Buffer` object for each send/receive operation is no longer required. The `iodev` device has been implemented using Java IO sockets, which support the direct communication of any serializable object. This implementation will allow the analysis of the impact of the buffering avoidance in MPJ Express, as well as the comparison of a Java IO socket implementation versus a Java NIO-based

DOI: 10.1002/cpe

one. Nevertheless, the use of Java IO sockets has required a significantly higher effort in developing scalable nonblocking communications, whose support is direct in Java NIO sockets but not in IO sockets. However, the IO socket API allows the use of our high-performance Java IO socket implementation, JFS (see Figure 2).

Therefore, the `iodev` device can use either JFS or JVM IO socket implementations, thus providing portability through the use of a JVM library (IO sockets) as well as high performance because it can rely on JFS on HPC hardware: 10 Gigabit Ethernet, SCI, Myrinet, and InfiniBand high-speed networks and shared memory architectures.

### *3.1. Communication operation in* iodev

The communication operation in `iodev` presents significant advantages such as (i) the removal of the dependence on the buffering layer, hence avoiding the overhead of the extra copies of the message data to the communication buffers; and (ii) the relying of `iodev` on the JVM garbage collection technology for memory management instead of on a custom implementation of buffering strategies used in previous communication devices (e.g., `niodev` and `mxdev`). However, the following disadvantages have also to be considered:

1. Primitive data type arrays have to be serialized as the Java NIO bulk get/put methods, which avoid this costly process, are not available for Java IO sockets.
2. An extra JNI copy has to be done between the data in the JVM and native memory in order to transfer the data, while Java NIO *direct* byte buffers avoid this copy as they reside in the OS memory and allow native code to access their data directly.
3. Java NIO buffers provide a standard and efficient support for MPJ derived data types, whereas in `iodev` this support must be implemented from scratch.

Nevertheless, the first two issues can be overcome with the use of the high-performance JFS library, which avoids the serialization of primitive data types arrays and the extra data copies through JNI. Thus, the serialization can be avoided through the use of a Java sockets extended API (see Listing 2) that allows the direct transfer of primitive data type arrays, even supporting the direct communication of portions of primitive data type arrays. As JVM sockets cannot send array portions (except for parts of byte arrays), a new array must be created to store the data to be serialized and then be sent. This costly process is repeated analogously at the receiver side.

As Java parameters are passed by value, the receiving methods (e.g., `irecv`) are unable to modify the receiving object reference. Thus, an intermediate structure (an array) is needed to store the reference to the read object and maintain this reference in the receiving process. This fact limits Java communication devices to receive only arrays.

The operation of the `iodev` communication mechanism starts checking if the message is a primitive data type array. In this case, if it is possible (e.g., if the sockets extended API is available), the serialization of the data is avoided. Otherwise, data has to be serialized and sent using JVM IO sockets, except when the message handled is an array of arrays, where each element will be processed independently. In this scenario, the communication method is recursively called for each element of the array, which can be either serialized or not.

Furthermore, in `iodev`, the message data have to be copied through JNI in order to be sent/received, unlike using Java NIO *direct* byte buffers whose data are directly accessible into

```
SocketOutputStream.write(int buf[], int offset, int length);
SocketOutputStream.write(double buf[], int offset, int length);
SocketOutputStream.write(float buf[], int offset, int length);
  ...
SocketInputStream.read(int buf[], int offset, int length);
SocketInputStream.read(double buf[], int offset, int length);
SocketInputStream.read(float buf[], int offset, int length);
  ...
```

Listing 2. Sockets extended API for communicating primitive data type arrays directly.

native communication operations. Nevertheless, it is possible to avoid these extra data copies through the use of native methods.

### *3.2. Implementation of the* iodev *communication device*

The `iodev` device implements nonblocking low-level communication primitives on top of Java IO sockets. In `iodev`, each process uses two TCP sockets, one for sending and another for receiving, in order to be connected to every other process. This design decision reduces synchronization overheads when sending/receiving data to/from the same peer process, while not restricting much the scalability of `iodev`, which is able to communicate with up to 500 peers by default. In fact, `iodev` is generally limited by the maximum number of open file descriptors (each socket connection has one open file descriptor) that is usually set to 1024. The access to these sockets has been synchronized with locks, both for reading and for writing, as several threads have read/write access to these sockets.

An `iodev` message consists of a header plus data. The message header includes the data type sent, the source identification, the message size, the tag, and the context and control information. In order to reduce the overhead of multiple accesses to the network, the `iodev` message header is buffered. Once the message header buffer has been filled in, it is written to the network. The message data are next sent to the network. Thus, only two accesses are required for each message. Although for very short messages (<4 KB), the header and data are merged in order to perform a single-socket write call. This optimization has been evaluated experimentally corroborating the benefits of this approach in terms of start-up latency overhead reduction. In fact, Figures 3–9 confirm that there is no evidence of performance penalties for `iodev` short messages. Moreover, when the source and the destination of a message are the same, the socket communication is replaced by an array copy.

In `iodev`, all communication methods are based on the nonblocking primitives `isend`/`irecv`. Thus, blocking communication methods are implemented as a nonblocking primitive followed by a nonblocking wait (`iwait`) call. In order to handle the nonblocking communications, their `Request` objects are internally stored in two disjoint sets of pending communication requests.

The message reception is carried out by both the input handler, a thread in charge of receiving data (also known in the literature as the *progress engine*), which is constantly running from the `init` up to the `finish` call, and the `Request.iwait` method. Usually, in message-passing libraries, both native and Java implementations, only the input handler receives messages. This, in order to continue the execution, presents a high overhead that consists of the following: first, the reception of the message by the input handler; second, the notification of the reception to the `Request` object (which is in a `wait` state); third, the waking up of this waiting object; and fourth, the context switching between the input handler and the `Request`. However, in `iodev`, both the input handler thread and the `Request.iwait` method receive messages. Thus, if `Request.iwait` receives the message, the overhead of the input handler reception is avoided.

The `iodev` device implements the `iwait` operation by means of a polling strategy together with periodically issued `yield` calls, which decrease `iwait` priority in order not to monopolize system CPU. This strategy allows significant reduction of message latency, especially in a scenario of undersubscription (e.g., running four processes on eight available cores) where both the user thread and the input handler can be polling simultaneously, in exchange for a moderate CPU overhead increase compared with the approach where only the input handler (progress engine) receives data. This approach allows `iodev` to obtain significant benefits, especially in communication-intensive codes, as message latency reduction provides higher scalability than the saving of some CPU cycles. However, when the number of processes is equal or higher than the number of available cores, the yield calls of the input handler are likely to reduce significantly its polling activity. Thus, the number of polling threads running at a given time would tend to be similar to the number of available cores. Finally, the use of two threads per process has also been evaluated in MPI [19], showing that this approach eliminates context switches, reduces scheduler overhead, and diminishes the overhead of privilege changes between user space and kernel space.

### *3.3.* iodev *communication protocols*

The iodev device implements two communication protocols, eager and rendezvous.

On the one hand, the eager protocol delivers the message data without waiting for the receiver to request it, on the assumption that the receiver has available storage space (otherwise, an out-of-memory exception is thrown). This direct communication is targeted to short messages, typically below 128–512 KB (configurable threshold). In fact, it minimizes the start-up latency (the 0-byte message latency), as no control message is required, although it adds the overhead of an extra copy when the receiver is not waiting for a particular message, and hence the communication will suffer the overhead of extra copies. In fact, in this latter scenario, the input handler or the Request.iwait method will temporarily receive the data, being copied later to the final destination.

On the other hand, the rendezvous protocol does not deliver the data until the receiver explicitly requests it, thus preventing the temporal storage of the messages whose corresponding receive operation has not been already called. The use of control messages in the implementation of this protocol makes it suitable for large messages, typically above 128–512 KB, although its communication strategy is sensitive to high start-up latencies as it implements a three-step protocol: (i) the source sends a ready-to-send message; (ii) the destination replies with a ready-to-receive message; and (iii) data are actually transferred. Thus, this strategy avoids the overhead of extra data copies, although it increases protocol overhead. However, its impact is usually reduced for large messages.

The benefits of these protocols on the performance of the applications can be significant. Thus, the eager protocol reduces the start-up latency, allowing Java applications with intensive short-message communications to increase their scalability. Moreover, the rendezvous protocol maximizes communication bandwidth, thus reducing the overhead of message buffering and network contention. Therefore, both protocols support the scalable performance of MPJ applications.

### *3.4. Integration of* iodev *in MPJ Express*

In order to take advantage of the iodev features (e.g., buffering avoidance and high-speed network support), MPJ Express has to implement the support for its extended API, which communicates regular objects instead of mpjbuf.Buffer objects. This task is relatively complex as the MPJ Express architecture is tightly bound to the buffering layer. In fact, the other low-level communication devices only support the communication of mpjbuf.Buffer objects, as shown in Listing 3 for the Send method. Here, the data are packed onto an mpjbuf buffer before being sent. The implementation of the receive operation is analogous.

The integration of iodev in MPJ Express requires a design that bypasses the buffering layer, which has meant the implementation of an alternative trunk version of MPJ Express, which replaces calls to the buffering layer and the xdev API with invocations to the iodev extended API. Thus, MPJ applications can rely on iodev, which can achieve better performance results, as will be shown in the performance evaluation (Sections 4 and 5).

```
void Send(Object buf, int offset, int count,
          Datatype datatype, ProcessID dstID, int tag) {
    int context = 0; //default value for point-to-point comms
    Packer packer = datatype.getPacker();
    mpjbuf.Buffer wBuffer = datatype.createWriteBuffer(count);
    packer.pack(wBuffer, buf, offset, count);
    wBuffer.commit();
    xdev.send(wBuffer, dstID, tag, context);
    wBuffer.clear();
    wBuffer.free();
}
```

Listing 3. Current MPJ Express Send method implementation.

The implementation of the `xdev.send` method in `iodev` (see Listing 4) entails the inspection of the handled message data. This is accomplished via the use of the Java reflection API. The first steps are composing the message header and checking if the data is an array of a primitive data type or an array of objects. Further steps include collecting more information about the data such as checking if the data is a multidimensional array or if it involves the communication of only a part of an array. Then, the message is written onto the socket using either JVM sockets or JFS methods, depending on the availability of this high-performance socket implementation. Its presence is detected through a typecasting against its own stream implementation, which will be successful only if the library is available. The implementation of the receiving operation follows a similar approach.

An important design decision taken into account in the integration of this communication device is the thread safety of the overall solution. This fact allows the efficient exploitation of hybrid shared/distributed memory architectures, such as multicore clusters, through the combination of

```
void send(Object data, int offset, int count, ProcessID dstID,
                       int tag, int context) {

  // synchronized access to the outputStream
  OutputStream outputStream = acquireStreamforDestination(dstID);

  if (outputStream instanceof jfs.net.SocketOutputStream) {
       jfsOutputStream = (jfs.net.SocketOutputStream) outputStream;
       usingJFS = true;
  }

  Class dataClass = data.getClass();
  if (dataClass.isArray()) {
      Class classComponent = dataClass.getComponentType();

      Util.writeHeader(outputStream,classComponent,offset,count,tag,context);

      if (classComponent.isPrimitive()) {
          if (usingJFS) jfsOutputStream.write(data,offset,count);
          else {
              if (classComponent != Byte.TYPE) {
                     //serialize data
                    outputStream.write(serialized_data);
              } else {
                    outputStream.write(data,offset,count);
              }
          }
      } else { // sending an array of objects
          Class innerClass;
          innerClass = java.lang.reflect.Array.get(data, offset).getClass();
          if (innerClass.isArray()) {
              for(i=0;i<count;i++) {
                  Object objRef = java.lang.reflect.Array.get(data, offset+i);
                  send(objRef, 0, objRef.length, dstID, tag, context+i);
              }
          } else {
              if(count < data.length) {
                    // serialize count elements to be sent
              } else {
                    // serialize data
              }
              outputStream.write(serialized_data);
           } //fi innerClass isArray?
        } //fi classComponent isPrimitive?
    } //fi dataClass isArray?

    outputStream.flush()
    releaseStreamforDestination(dstID,outputStream);
 }
```

Listing 4. Proposed `xdev.send` pseudocode with buffering avoidance (eager protocol).

DOI: 10.1002/cpe

message passing and the built-in multithreading support of Java. Thus, MPJ Express maintains its MPI_THREAD_MULTIPLE thread-safety level. This means that multiple threads can communicate without restriction, taking advantage of the increasing number of cores available per system.

Once the avoidance of the buffering layer has been implemented in the MPJ Express library, it is necessary to evaluate the communication device presented here, `iodev`, together with its impact on the overall performance of MPJ applications.

## 4. PERFORMANCE EVALUATION

This work presents a performance evaluation of MPJ Express communication devices (`iodev`, `niodev`, `mxdev`, and `smpdev`) compared with native MPI libraries and mpiJava [20, 21], an MPJ wrapper implementation on top of MPI, on Gigabit Ethernet, Myrinet, SCI, and InfiniBand networks, as well as on a shared memory multicore scenario. Thus, this section includes a microbenchmarking of point-to-point primitives, using our own MPJ microbenchmarking suite [22], and Section 5 presents an analysis of the impact of the communication devices on the scalability of two representative message-passing applications, finite-difference time-domain (FDTD) and Gadget [18].

### *4.1. Experimental configuration*

Two multicore clusters have been used in this performance evaluation of Java message-passing communication devices.

The first system is a cluster of eight dual-processor x86_64 nodes (Intel Pentium 4 Xeon 5060 dual core at 3.2 GHz, 4 GB of memory [Intel Corp., Santa Clara, CA, USA]) interconnected via SCI (D334 network interface Dolphin Interconnect Solutions, Oslo, Norway), Myrinet (‘F’ Myrinet 2000 card [Myricom, Inc., Arcadia, CA, USA]), Gigabit Ethernet (Intel PRO/1000 [Intel Corp., Santa Clara, CA, USA]), and InfiniBand (QLogic IBA7220 4x Double Data Rate (DDR) [Aliso Viejo, CA, USA], 16 Gbps). Each node has four cores, allowing the evaluation of shared memory communications.

Regarding the software configuration of this testbed, the OS is Linux CentOS 5.1 with kernel 2.6.18 and C compiler Intel icc 11.0 (used with -O3 flag). The JVM is the Sun Java Development Kit 1.6. The native communication libraries (see Figure 2) are the SCI Sockets 3.1.4, Dolphin Interconnect Solutions (DIS) 3.1.11 (it includes Interconnect Resource Manager, Software Infrastructure for SCI and SCILib) and SCIP 1.2.0 on SCI; MX 1.1.1 and Sockets-MX 1.1.0 on Myrinet; and Sockets Direct Protocol, IPoIB (the IP emulation over InfiniBand) and InfiniBand Verbs (IBV), from the Open Fabrics Enterprise Distribution (OFED) drivers 1.4, on InfiniBand.

The second system used in this performance evaluation is the Finis Terrae supercomputer [23], an InfiniBand cluster with 2400 cores (14 TFlops). This supercomputer is also the system selected for the analysis of performance scalability of MPJ applications (shown in Section 5), because of its high number of cores.

The Finis Terrae consists of 142 Hewlett–Packard (HP) Integrity rx7640 nodes (Hewlett–Packard Company, Palo Alto, CA, USA), each of them with 16 Montvale Itanium 2 (IA64) cores (Intel Corp.) at 1.6 GHz and 128 GB of memory. The InfiniBand NIC is a 4X DDR Mellanox MT25208 (Mellanox Technologies, Inc., Sunnyvale, CA, USA) (16 Gbps). The OS is SUSE Linux Enterprise Server 10 with C compiler Intel icc 10.1.012 (used with the -O3 flag). The JVM is BEA JRockit 5.0 (R27.5), selected because of its high performance on this system, outperforming the other JVM available for IA64 systems, the Sun JVM. The native communication library is OFED version 1.3.

The Java message-passing communication devices evaluated are `niodev`, `mxdev`, and `smpdev` from MPJ Express 0.36 and an internal release of `iodev` (with JFS 0.3.1). Additionally, for comparison purposes, the following native MPI libraries have been evaluated: Scali’s MPI (ScaMPI) 1.13.8 on SCI; MPICH-MX 1.2.6 on Myrinet; Intel MPI 3.2.0.011 on InfiniBand, Gigabit Ethernet, and shared memory (all of them on the x86_64 cluster); and HP MPI 2.2.5.1 on InfiniBand and shared memory on the Finis Terrae. Moreover, mpiJava 1.2.5x has also been benchmarked. The experimental results have been obtained both at the device level (`xdev`) and at the MPJ API level (see Figure 1) for the communication devices evaluated, iodev, niodev, mxdev, and smpdev, whereas the native MPI libraries have been benchmarked only at the MPI API level (this high-level layer usually adds quite low overhead, almost negligible, on its underlying low-level message-passing devices). The mpiJava library has been benchmarked at the MPJ API level as it relies on native MPI libraries for communication, instead of on Java message-passing communication devices.

### *4.2. Point-to-point microbenchmarking on the x86_64 cluster*

Figures 3–7 show the measured point-to-point latencies and bandwidths on the x86_64 cluster when communicating byte arrays on Gigabit Ethernet (using niodev and iodev), SCI (using niodev and iodev), Myrinet (using mxdev and iodev), InfiniBand (using niodev and iodev), and shared memory (using smpdev and iodev).

An analysis of the performance results reveals the following:

1. iodev shows similar performance at the communication device level (xdev) and at the MPJ layer, MPJ(iodev), as it does not incur any costly operation at the MPJ layer. Thus, for clarity purposes, their performance is shown in Figures 3–7 as MPJ(iodev)/iodev.
2. niodev, mxdev, and smpdev present, however, different performance between MPJ and xdev levels as the operation of these devices is tightly coupled with the MPJ Express buffering layer, incurring significant overhead at the MPJ level (the packing and unpacking of data is done at this level). This overhead for short messages is around 20 $\mu$s for niodev, 10 $\mu$s for mxdev (half of niodev overhead thanks to its native buffer handling), and 3 $\mu$s for smpdev. However, this byte array packing overhead is slightly lower than using derived and other primitive data types.
3. MPJ level buffering overhead (incurred by niodev, mxdev, and smpdev) is also significant for large messages, reducing communication performance down to a half, especially on InfiniBand and shared memory scenarios, which provide high raw bandwidth. However, when the NIC is the main performance bottleneck, as for Myrinet and Gigabit Ethernet, the performance decrease is much less important, less than 15%.
4. As a direct consequence of the reduction of the serialization and buffering overheads, MPJ(iodev) obtains the best performance among current MPJ communication devices.

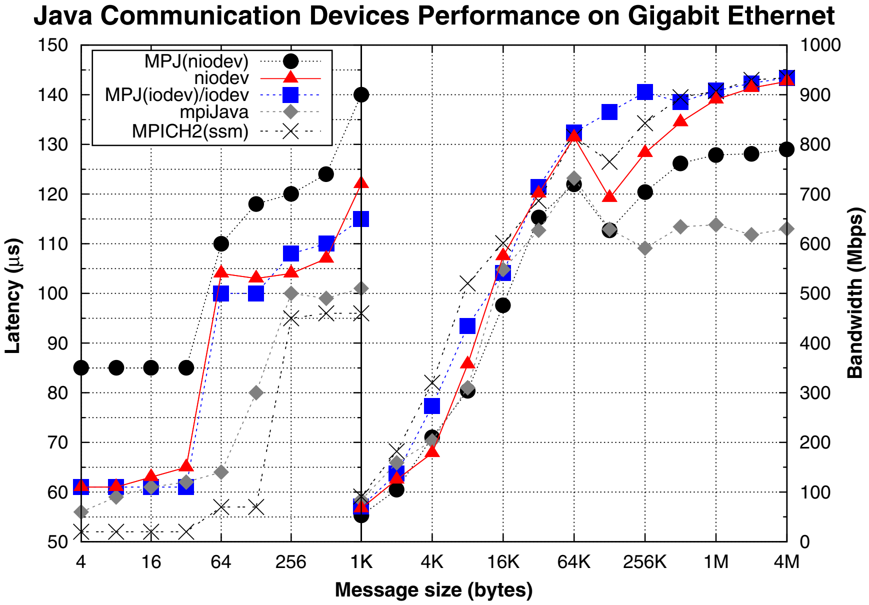

Figure 3. Message-passing point-to-point performance on Gigabit Ethernet (x86_64 cluster). MPJ, Message Passing in Java; MPICH2(ssm), MPICH2 (sockets and shared memory).

Additionally, a general comparison of Java communication devices against mpiJava and native MPI results shows the following:

1. MPI libraries generally obtain the best performance, as their implementations are more developed, extended, and mature than those of the MPJ ones. In fact, their high-speed networks and shared memory support is more efficient, although the thread-based approach of `smpdev` can outperform MPI on shared memory.
2. mpiJava achieves the lowest Java start-up latencies thanks to a reduced wrapping overhead. In fact, it shows results around 10 $\mu$s higher than the underlying MPI library, independently of the MPI library and the underlying HPC hardware (high-speed network or shared memory).
3. mpiJava shows poor large-message performance due to the overhead of the JNI copy of the message data between the Java heap and its underlying MPI library (which even results in a higher overhead than the data copy performed in `niodev`).

The network is the main performance bottleneck for the Gigabit Ethernet results (see Figure 3), limiting its bandwidth to a maximum of 1000 Mbps, while showing start-up latencies quite high (more than 50 $\mu$s), imposing poor performance for short messages. Moreover, the interrupt handling scheme of the Gigabit Ethernet Linux driver (interrupt coalescence) delays communications, causing latencies to be around multiples of 50 $\mu$s (driver notifications are handled every 50 $\mu$s, approximately). Thus, `MPJ(iodev)/iodev` performance is close to that of MPI, especially for large messages, outperforming `MPJ(niodev)` and especially mpiJava thanks to the buffering avoidance.

Figure 4 shows the results of the evaluated libraries on SCI, where the native MPI implementation obtains the lowest start-up latency, 4 $\mu$s, followed by mpiJava, with 13 $\mu$s (a 9 $\mu$s start-up latency overhead over the underlying ScaMPI), and `iodev`, which achieves a similar result, 14 $\mu$s. Nevertheless, `niodev` shows poorer short-message performance, 47 $\mu$s start-up latency. Finally, `MPJ(niodev)` shows the highest latency (71 $\mu$s) due to the buffering overhead.

Regarding large-message bandwidths, `MPJ(iodev)/iodev` presents the best performance among Java communication devices, achieving similar performance to the native MPI library, ScaMPI. Compared with mpiJava, `MPJ(iodev)` presents similar bandwidths for messages up to 64 KB; however, for longer messages, mpiJava performance falls below 1500 Mbps. Finally, the high start-up latency of `niodev` and `MPJ(niodev)` has a significant impact on performance.

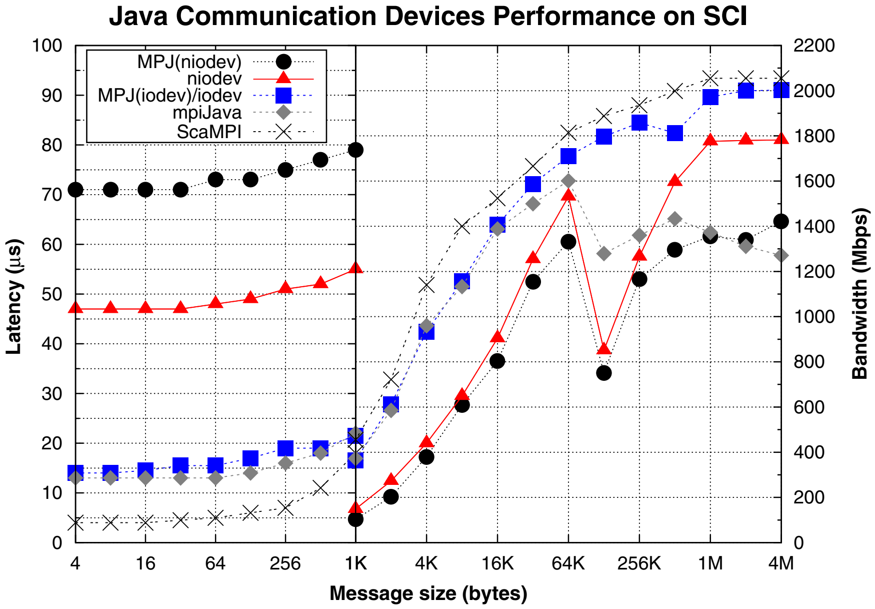

Figure 4. Message-passing point-to-point performance on Scalable Coherent Interface (SCI) (x86_64 cluster). MPJ, Message Passing in Java; ScaMPI, Scali Message-Passing Interface.

DOI: 10.1002/cpe

Thus, messages sent through the rendezvous protocol (message size >128 KB), which involves two control messages and the actual data transfer (hence, three data transfers per message), incurs a poor performance, especially for 128- and 256-KB message sizes, obtaining less than half of `iodev` performance on this scenario (see Figure 4).

Figure 5 shows point-to-point performance results on Myrinet, where `MPJ(iodev)/iodev` start-up latency is 17 $\mu$s, which is lower than `MPJ(mxdev)` (24 $\mu$s) and similar to mpiJava and `mxdev` (13 and 16 $\mu$s, respectively). Here, the start-up buffering overhead, which is the difference between `MPJ(mxdev)` and `mxdev` start-up latency, is lower than the difference between `MPJ(niodev)` and `niodev` as the communication is handled by native methods. Moreover, the Myrinet 2000 network, with a theoretical maximum bandwidth of 2000 Mbps, is the main performance bottleneck for large messages, limiting `MPJ(iodev)` large-message bandwidth to 1800 Mbps, whereas `MPJ(mxdev)`, which additionally incurs a significant MPJ buffering overhead, obtains results around 1300 Mbps.

Figure 6 shows the performance results on InfiniBand, where start-up latencies are higher than on SCI and Myrinet. Moreover, large-message bandwidths, although higher than on SCI and Myrinet, are far from the theoretical limit, 16 Gbps, due to the communication protocol processing overhead (this analysis is supported by the InfiniBand evaluation on the Finis Terrae, presented in Subsection 4.3). The analysis of the particular performance results on InfiniBand confirms the conclusions derived from previous results (Figures 3–5) that MPI obtains the best performance, followed by `MPJ(iodev)/iodev` when sending large messages and by mpiJava for short messages. Finally, `MPJ(niodev)` and `niodev` show poor start-up latencies and high processing overhead, which penalizes especially MPJ large-message performance, whose results are below 2.2 Gbps.

The increasing number of cores per system heightens the need for efficient message-passing communications on shared memory. The performance evaluation on our testbed (Figure 7) shows lower start-up latencies compared with high-speed networks, as well as higher bandwidths. Here, MPI usually obtains the best performance, although `MPJ(iodev)/iodev` can achieve quite competitive results thanks to the efficient communications support of its underlying layer (JFS over UNIX sockets). Regarding `MPJ(smpdev)`, although thread-based intra-process transfers can present the highest transfer rates, its performance is severely limited by synchronization and buffering overheads, thus showing poorer performance than `MPJ(iodev)/iodev`. Additionally, mpiJava obtains

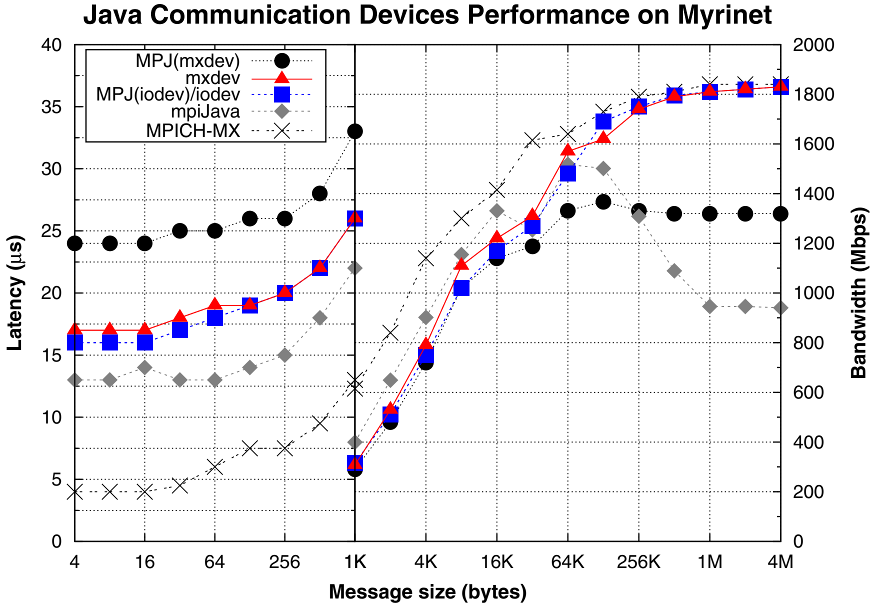

Figure 5. Message-passing point-to-point performance on Myrinet (x86_64 cluster). MPJ, Message Passing in Java; MPICH-MX, MPICH-Myrinet Express.

DOI: 10.1002/cpe

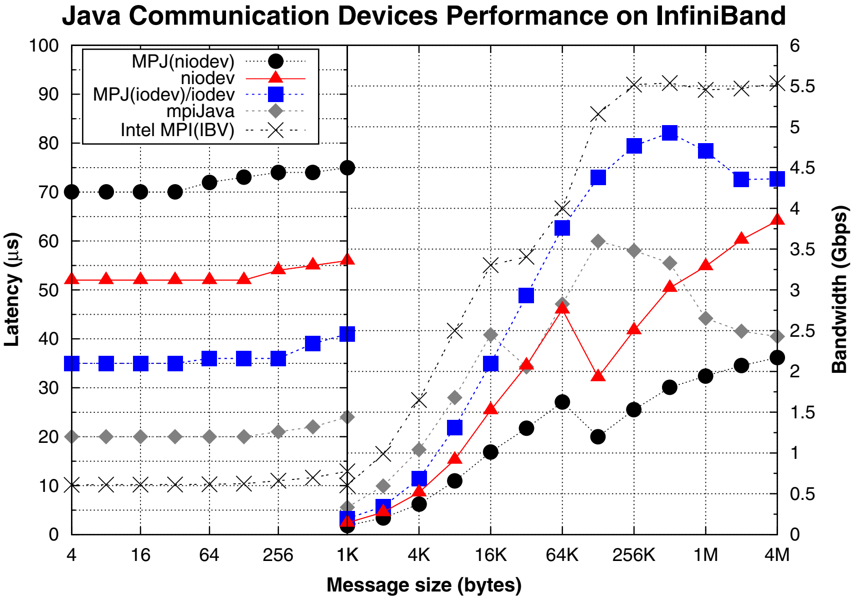

Figure 6. Message-passing point-to-point performance on InfiniBand (x86_64 cluster). MPJ, Message Passing in Java; MPI(IBV), Message-Passing Interface (InfiniBand Verbs).

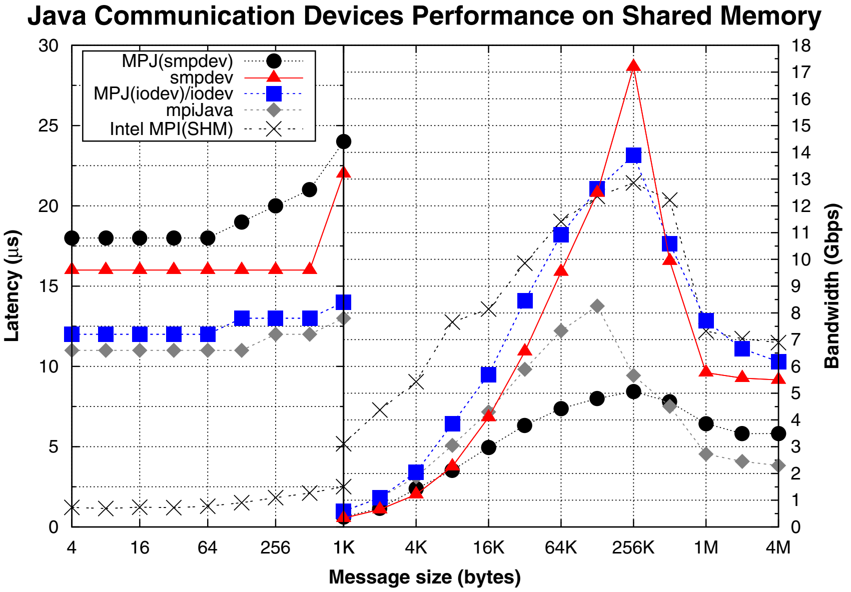

Figure 7. Point-to-point performance on shared memory (x86_64 cluster). MPJ, Message Passing in Java; MPI(SHM), Message-Passing Interface (shared memory).

good start-up latencies but poor large-message performance. Moreover, it is noticeable that the performance of `iodev` and `smpdev` drops for 512 KB, the shortest message using the rendezvous protocol in this scenario, as this protocol involves three communications steps instead of only one (as in the eager protocol). Thus, `iodev` and `smpdev` obtains higher performance for a 256-KB message (eager protocol) than for a 512-KB message (rendezvous protocol). Finally, the performance of all these libraries also drops when the total dataset plus the associated auxiliary storage exceeds the L2 cache size (the Xeon 5060 has a 2-MB L2 cache, and from 1-MB message size, the data do not fit in cache).

DOI: 10.1002/cpe

### *4.3. Point-to-point microbenchmarking on the Finis Terrae*

Figures 8 and 9 show latencies and bandwidths of point-to-point operations on the Finis Terrae, using InfiniBand and shared memory communications, respectively. The motivation for this benchmarking is the analysis of the MPJ Express devices on a supercomputer, a high-end environment whose hardware provides higher communications performance. Additionally, this system has been used for evaluating the scalability of representative message-passing applications, so the characterization of the point-to-point performance is of special interest for the analysis of their results (presented in Section 5).

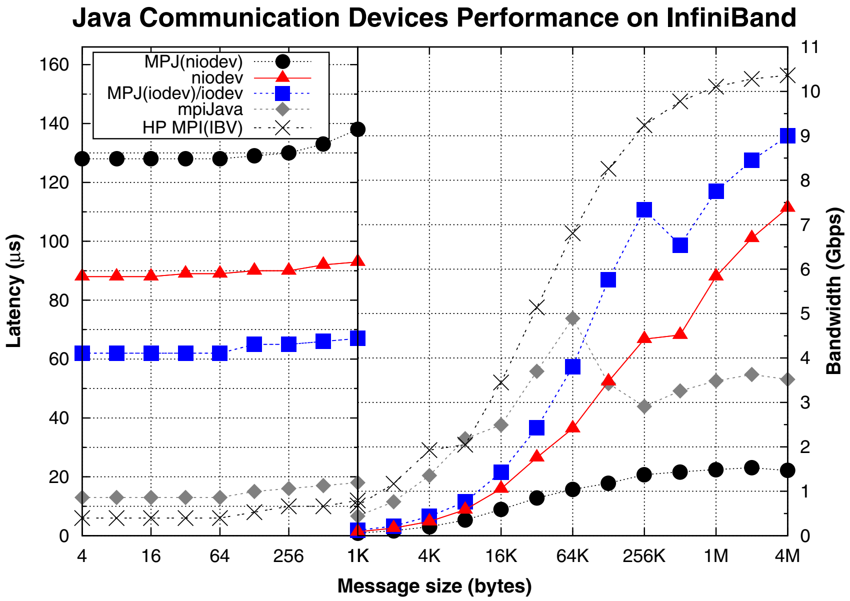

Figure 8. Point-to-point performance on InfiniBand (Finis Terrae). MPJ, Message Passing in Java; HP MPI(IBV), Hewlett–Packard Message-Passing Interface (InfiniBand Verbs).

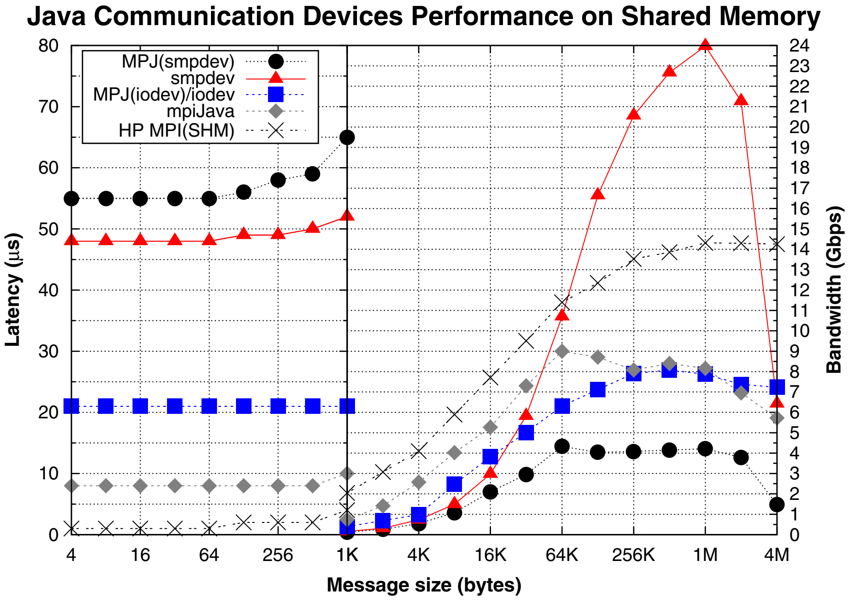

Figure 9. Point-to-point performance on shared memory (Finis Terrae). MPJ, Message Passing in Java; HP MPI(SHM), Hewlett–Packard Message-Passing Interface (shared memory).

DOI: 10.1002/cpe

The microbenchmarking on the Finis Terrae has shown, compared with the x86_64 cluster results, higher performance differences between Java (niodev and iodev) and native code (MPI), especially for short messages. The reasons for this higher gap are the relatively poor performance of the JVM on Linux IA64 systems, as well as the high performance of MPI on this supercomputer due to better hardware characteristics (e.g., performance of the processor–NIC connection and memory access performance). Thus, MPI shows start-up latencies as low as 6 $\mu$s on InfiniBand and below 1 $\mu$s on shared memory and large-message bandwidths above 10 Gbps on both scenarios. As a direct consequence of the higher performance of MPI, mpiJava, wrapping the MPI library, no longer shows the poorest large-message performance, outperforming MPJ on both shared memory and InfiniBand.

Figure 8 presents InfiniBand performance results on the IA64 supercomputer. The start-up latency of the Java communication devices on this system is quite high (62 and 88 $\mu$s for iodev and niodev, respectively) because of the poor performance of the JVMs on IA64 architectures. However, as the message size grows, their performance increases, obtaining bandwidths of up to 7.5 and 9 Gbps for niodev and iodev, respectively, whereas MPI obtains up to 10.4 Gbps. Thus, Java achieves up to 87% of the native communication performance (using iodev). Furthermore, the impact of the poor start-up latency is also noticeable in the performance of the rendezvous protocol for message sizes slightly higher than the protocol threshold, which is set to 256 KB, as this protocol involves three communication steps, two of them are control messages, that is, without actual message transfer but suffering the poor start-up latency. Thus, iodev obtains higher performance for a 256-KB message (eager protocol) than for a 512-KB message (rendezvous protocol).

Figure 9 shows shared memory performance of the evaluated communication libraries. Here, the most noticeable result is that smpdev achieves the highest performance for the message range (128 KB–2 MB), as it is the only communication device that performs intra-process communication, transferring data between two threads, whereas the remaining libraries perform inter-process data transfers. However, smpdev shows the poorest start-up latency due to its high synchronization overhead. This overhead limits smpdev short-message performance (up to 128 KB). Additionally, the MPJ buffering overhead also limits severely the performance benefits of MPJ(smpdev) for large-message communications, obtaining the poorest performance. In this scenario, iodev results are limited by the performance of UNIX sockets, its underlying communication mechanism that obtains approximately half of the performance of HP MPI, which relies on its SHared Memory (SHM) device.

## 5. IMPACT OF JAVA COMMUNICATION DEVICES ON HIGH-PERFORMANCE COMPUTING APPLICATIONS PERFORMANCE

This section presents the performance evaluation of two representative parallel applications, FDTD and Gadget [18], implemented using MPI (C) and MPJ and FDTD and Gadget [18]. The experimental results have been obtained on the Finis Terrae supercomputer (see Subsection 4.1), which allows the evaluation of up to 64 cores on a multicore shared memory machine (an Itanium 2 Montvale-based ccNUMA Integrity Superdome node [Hewlett–Packard Company] within the Finis Terrae). Moreover, it allows the performance evaluation on distributed memory, using InfiniBand as interconnection network and running the applications with four cores per node and up to 64 nodes (e.g., a 32-core execution involves eight nodes and four cores per node). This latter configuration experimentally obtains the best performance results. In fact, the use of a higher number of cores per node (more than four) turns the interconnection network into a major performance bottleneck. The analysis of the results of this evaluation confirms that the research on Java communication devices has a highly positive impact on the scalability of these applications.

### *5.1. Finite-difference time-domain performance evaluation*

A message-passing (MPI/MPJ) version [18] of the FDTD method [24] (widely used in electromagnetism) has been evaluated. This parallel FDTD application divides the workload equally among the available computational resources, requiring frequent updates from the neighbor processes state during the simulation. These updates are implemented using nonblocking point-to-point data transfers. Moreover, the Java FDTD code has been optimized, avoiding performance bottlenecks such as the use of multidimensional arrays [25].

Figure 10 presents the execution times and scalability of MPI and MPJ FDTD when simulating 200 steps on 4096 × 4096 grids. The exploitation of the data locality explains the speedup increase observed for a number of cores of 64 and higher, when the dataset fits entirely in cache (Itanium 2 Montvale 9140 has 9-MB L3 capacity per core).

Regarding shared memory results (only available up to 64 cores), `MPI(SHM)` shows high speedups, whereas MPJ presents poor scalability for a number of cores of 16 and higher, not taking advantage both MPJ shared memory solutions of this ccNUMA architecture, especially when using `MPJ(smpdev)` as it suffers from important synchronization overheads, a significant performance bottleneck for an application with frequent short-message transfers.

Although the scalability of MPJ when using InfiniBand is higher than that of MPI, the results are not directly comparable, as the sequential runtimes of the C and Java FDTD codes are different (80 and 179 seconds, respectively), and hence their parallel execution times. The reason for this noticeable gap is the poor performance of the JVMs on Linux IA64 systems, which allows higher speedups. This reinforces one of the main conclusions of this paper, that MPJ can help bridge the gap between C and Java applications in HPC, especially when using `iodev` on InfiniBand, thanks to the avoidance of the buffering overhead and the efficient exploitation of high-speed networks.

### 5.2. *Gadget performance evaluation*

Gadget [26, 27] is a very popular application in cosmology simulation. The parallelization strategy, both with MPI (C) and MPJ [18], is an irregular and dynamically adjusted domain decomposition, with intensive communication between processes.

Figure 11 presents Gadget performance results for a galaxy cluster formation simulation with two million particles in the system (simulation available within the Gadget code and examples bundle). As Gadget is a communication-intensive application, the speedups obtained on the Finis Terrae supercomputer are below 50.

Regarding the results on shared memory, on the one hand, MPJ, especially with `smpdev`, shows the poorest scalability. On the other hand, `MPI(SHM)` obtains the highest speedups when using up

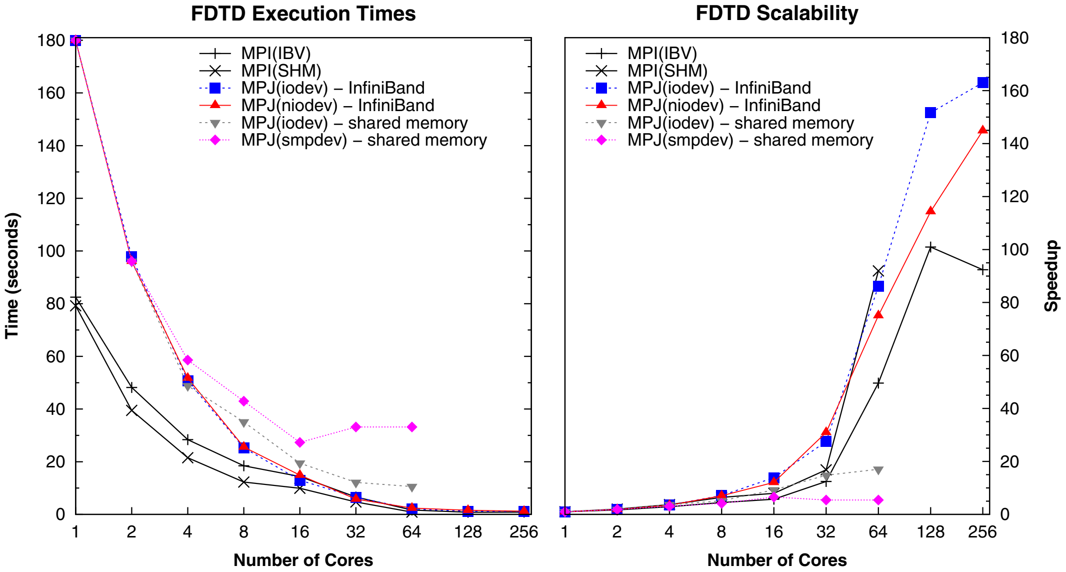

Figure 10. Execution times and scalability of Message-Passing Interface/Message Passing in Java finite-difference time-domain (MPI/MPJ FDTD) on the Finis Terrae. IBV, InfiniBand Verbs; SHM, shared memory.

DOI: 10.1002/cpe

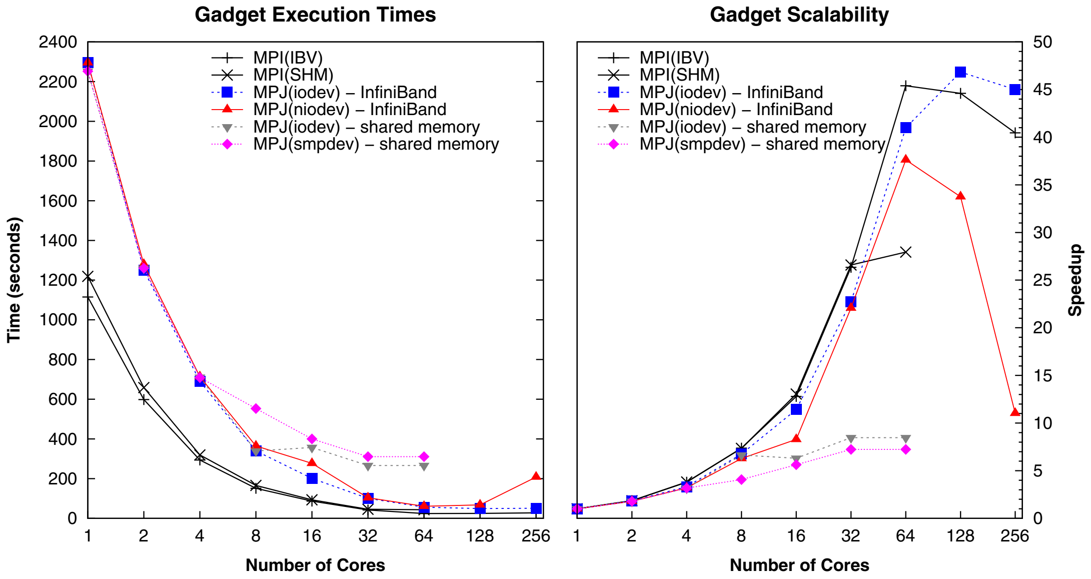

Figure 11. Execution times and scalability of Message-Passing Interface/Message Passing in Java (MPI/MPJ) Gadget on the Finis Terrae. IBV, InfiniBand Verbs; SHM, shared memory.

to 32 cores, closely followed by `MPI(IBV)`. Although the `smpdev` device obtains reasonably good point-to-point performance, the scalability of `MPJ(smpdev)`, which motivates the optimization of this device with a special focus on reducing the synchronization overhead, is poor.

The scalability achieved by MPJ with the new `iodev` device on InfiniBand is similar to that of MPI, but they can not be compared directly as the sequential runtimes of the C and Java Gadget codes are quite different (1114 and 2295 s, respectively), and hence their parallel execution times. In fact, although the MPJ Gadget implementation obtains good scalability, its runtimes are still higher than the MPI ones.

## 6. RELATED WORK

The use of pluggable low-level communication devices is widely extended in message-passing libraries [28]. Thus, MPICH [29] includes several devices that implement the Abstract Device Interface (ADI), a low-level messaging API, on several communication layers such as high-speed networks (e.g., Myrinet and InfiniBand) and shared memory. Moreover, Open MPI [30] also supports high-speed interconnects and shared memory through the implementation of Byte-Transfer-Layer (BTL) communication devices for each communication technology (e.g., Myrinet and InfiniBand). Regarding MPJ libraries, MPJ Express [6] also follows this approach with the `xdev` layer [14], providing communication devices for different interconnection technologies (Java NIO sockets, MX for Myrinet, and shared memory). Other MPJ libraries [2] (e.g., P2P-MPI and Jcluster) only provide support for Java NIO and/or IO sockets, without taking into account either high-speed networks or shared memory systems. The only exception is the MPJ/Ibis, which is discussed next.

Since the introduction of Java, there have been several implementations of Java messaging libraries for HPC [2]. These libraries were initially implemented using Java Remote Method Invocation (RMI), but its TCP/IP default implementation presents quite high overhead, especially the data marshalling and transport mechanisms. Thus, the optimization of the RMI protocol has been the goal of several projects, such as KaRMI [31], RMIX [32], Manta [33], Ibis RMI [34], and Opt RMI [35]. However, the use of nonstandard APIs, the lack of portability, and their high overhead, which is still significantly larger than socket latencies, have restricted their applicability. Therefore, although Java communication middleware (e.g., message-passing libraries) used to be based on RMI, current Java communication libraries are implemented by (i) wrapping an underlying native messaging library like MPI through JNI or by (ii) using Java sockets. The use of JNI presents portability, security, and dependency issues but in exchange of usually a higher performance, thanks to taking advantage of the efficient communications of native MPI libraries. The other approach, which is the use of a low-level API, Java sockets, requires an important programming effort, especially for providing scalable solutions. However, both solutions are able to provide higher throughput, key in HPC.

An example of a wrapper library is mpiJava [20, 21], which usually achieves high performance but presents some portability issues as it only supports some native MPI libraries, as wrapping a wide number of functions and heterogeneous runtime environments entails an important maintaining effort. Additionally, this implementation is not thread safe, being unable to take advantage of multicore systems through multithreading.

Socket-based MPJ libraries that support different communication technologies include, apart from our MPJ Express project, MPJ/Ibis [36] and P2P-MPI [37]. MPJ/Ibis is an MPJ implementation on top of Ibis [34], a parallel and distributed Java computing framework. Ibis can use both the 'pure' Java communications and the Myrinet high-speed network. There are two low-level communication devices in Ibis: TCPIbis, based on Java IO sockets, and NIOIbis, providing blocking and nonblocking communications through Java NIO sockets. Nevertheless, MPJ/Ibis is not thread safe, that is, it only provides blocking communications, and its Myrinet support is based on GM, an out-of-date low-level library on Myrinet, which has been superseded by MX and which is supported by most MPI libraries and MPJ Express. P2P-MPI is a 'pure' Java message-passing implementation whose communications are implemented on top of either Java IO sockets or NIO sockets. As this project is tailored to grid computing systems, it is focused on fault tolerance and dynamic discovery of computing resources.

However, the performance of socket-based MPJ libraries usually suffers from buffering [17] and serialization [38] overheads. In order to reduce their impact, several efforts have been devoted to optimize MPJ communications. Thus, our related project Fast MPJ [39] has served us to evaluate an early prototype of the `iodev` device within an MPJ library without buffering layer. Fast MPJ is a research implementation oriented to evaluate the scalability of new communication strategies. Jcluster [40] is an MPJ implementation that uses a reliable protocol based on UDP communications instead of TCP. Moreover, Java Object-Passing Interface (JOPI) [41] supports object communication through an MPI-like interface. However, its high communication overhead restricts its application to coarse-grain parallelism. Furthermore, the Parallel Java (PJ) project [42] is focused on hybrid shared memory/message-passing programming. Although these three latter projects, Jcluster, JOPI, and PJ, target programmability, they require the use of their own APIs and lack high-speed networks support, which has severely limited their adoption in the Java HPC arena. Finally, there have been several projects on serialization overhead reduction [31, 43].

## 7. CONCLUSIONS AND FUTURE WORK

The scalability of Java message-passing applications in HPC relies heavily on the performance of Java communication devices. As the buffering is among their main performance bottlenecks, several efforts have been carried out in order to reduce the overhead associated with its use in communications. Thus, the support for direct object communications, avoiding the use of buffers, has been implemented in a new communication device, `iodev`, implemented using Java IO sockets. This communication device has considered several protocols for a more efficient support of high-speed networks and shared memory systems, and it has reduced the nonblocking communications overhead. Moreover, this device can take advantage of efficient Java communication middleware, such as high-performance Java sockets, which implement several techniques to reduce serialization overhead. The performance evaluation of this device on Gigabit Ethernet, SCI, Myrinet, InfiniBand, and shared memory/multicore clusters has shown significant performance benefits, especially when the HPC hardware allows high-speed transfers, and the communication protocol is the main performance penalty, such as on shared memory and InfiniBand. Moreover, the scalability of MPJ applications can benefit from the reduction of the buffering overhead and the use of a more efficient high-speed network support. In fact, the development of efficient Java communication devices is bridging the gap between Java and native (compiled) applications in HPC.

Therefore, the increasing adoption of Java by the HPC community can take advantage not only from the built-in multithreading, security, portability, and higher programmability of Java but also from a more efficient MPJ communications support, in order to achieve higher productivity in parallel programming for multicore systems.

Further information, additional documentation, and software downloads of this project are available from the MPJ Express Project webpage http://mpj-express.org.

## ACKNOWLEDGEMENTS

This work was funded by the Ministry of Science and Innovation of Spain under Project TIN2010-16735. We gratefully thank CESGA (Galicia Supercomputing Center, Santiago de Compostela, Spain) for providing access to the Finis Terrae supercomputer.

## REFERENCES

1. Carpenter B, Chang Y, Fox G, Li X. Java as a language for scientific parallel programming. *Proceedings of the 10th International Workshop Languages and Compilers for Parallel Computing (LCPC '97)*, Minneapolis, MN, USA, 7–9 August 1997; 340–354.
2. Taboada GL, Touriño J, Doallo R. Java for high performance computing: assessment of current research and practice. *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java (PPPJ '09)*, Calgary, Canada, 27–28 August 2009; 30–39.
3. Java Grande Forum. Available from: http://www.javagrande.org [accessed on April 2011]
4. Blount B, Chatterjee S. An evaluation of Java for numerical computing. *Scientific Programming* 1999; **7**(2):97–110.
5. Dongarra J, Gannon D, Fox G, Kennedy K. The impact of multicore on computational science software. *CTWatch Quarterly* 2007; **3**(1):1–10.
6. Shafi A, Carpenter B, Baker M. Nested parallelism for multi-core HPC systems using Java. *Journal of Parallel and Distributed Computing* 2009; **69**(6):532–545.
7. Message Passing Interface Forum. Available from: http://www.mpi-forum.org [accessed on April 2011]
8. Godoy WF, DesJardin PE. On the use of flux limiters in the discrete ordinates method for 3D radiation calculations in absorbing and scattering media. *Journal of Computational Physics* 2010; **229**(9):3189–3213.
9. Padial-Collins NT, VanderHeyden WB, Zhang DZ, Dendy ED, Livescu D. Parallel operation of CartaBlanca on shared and distributed memory computers. *Concurrency and Computation: Practice and Experience* 2004; **16**(1):61–77.
10. Baker M, Carpenter B, Shafi A. MPJ Express meets Gadget: towards a Java code for cosmological simulations. *Proceedings of the 13th European PVM/MPI Users' Group Meeting (EuroPVM/MPI '06)*, Bonn, Germany, 17–20 September 2006; 358–365.
11. Darriba D, Taboada GL, Doallo R, Posada D. ProtTest 3: fast selection of best-fit models of protein evolution. *Bioinformatics* 2011; **27**(8):1164–1165.
12. Turner A. MoSeS Project. Available from: http://www.geog.leeds.ac.uk/people/a.turner/projects/MoSeS/ [accessed on April 2011]
13. Lim SB, Carpenter B, Fox G, Lee HK. A low-level communication library for Java HPC. *Proceedings of the 6th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP '05)*, Melbourne, Australia, 2–3 October 2005; 429–434.
14. Baker M, Carpenter B, Shafi A. A pluggable architecture for high-performance Java messaging. *IEEE Distributed Systems Online* 2005; **6**(10):1–4.
15. Shafi A, Manzoor J, Hameed K, Carpenter B, Baker M. Multicore-enabling the MPJ Express messaging library. *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ '10)*, Vienna, Austria, 15–17 September 2010; 49–58.
16. 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.
17. Baker M, Carpenter B, Shafi A. An approach to buffer management in Java HPC messaging. *Proceedings of the 6th International Conference on Computational Science (ICCS '06)*, Reading, UK, 28–31 May 2006; 953–960.
18. 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.
19. Hoefler T, Lumsdaine A. Message progression in parallel computing—to thread or not to thread? *Proceedings of the 10th IEEE International Conference on Cluster Computing (CLUSTER'08)*, Tsukuba, Japan, 29 September–1 October 2008; 213–222.

DOI: 10.1002/cpe

20. Baker M, Carpenter B, Fox G, Ko S, Lim SB. mpiJava: an object-oriented Java interface to MPI. *Proceedings of the 1st International Workshop on Java for Parallel and Distributed Computing (IWJPDC '99)*, San Juan, Puerto Rico, 12–16 April 1999; 748–762.
21. The mpiJava Project. Available from: http://www.hpjava.org/mpiJava.html [accessed on April 2011]
22. Taboada GL, Touriño J, Doallo R. Performance analysis of Java message-passing libraries on Fast Ethernet, Myrinet and SCI clusters. *Proceedings of the 5th IEEE International Conference on Cluster Computing (CLUSTER '03)*, Hong Kong, China, 1–4 December 2003; 118–126.
23. Finis Terrae Supercomputer. Available from: http://www.cesga.es/content/view/917/115/lang,en/ [accessed on April 2011]
24. Taflove A, Hagness S. *Computational Electrodynamics: The Finite-Difference Time-Domain Method* (3rd edn). Artech House Publishers: Norwood, MA, USA, 2005.
25. Moreira JE, Midkiff SP, Gupta M, Artigas PV, Snir M, Lawrence RD. Java programming for high-performance numerical computing. *IBM Systems Journal* 2000; **39**(1):21–56.
26. Springel V. The cosmological simulation code GADGET-2. *Monthly Notices of the Royal Astronomical Society* 2005; **364**(4):1105–1134.
27. Springel V, White SDM, Jenkins A, Frenk CS, Yoshida N, Gao L, Navarro J, Thacker R, Croton D, Helly J, Peacock JA, Cole S, Thomas P, Couchman H, Evrard A, Colberg J, Pearce F. Simulations of the formation, evolution and clustering of galaxies and quasars. *Nature* 2005; **435**(7042):629–636.
28. Pedroso H, Silva JG. An architecture for using multiple communication devices in an MPI library. *Proceedings of the 8th International Conference on High Performance Computing and Networking Europe (HPCN '00)*, Amsterdam, The Netherlands, 8–10 May 2000; 688–697.
29. Gropp W, Lusk E, Doss N, Skjellum A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing* 1996; **22**(6):789–828.
30. Open MPI Project. Available from: http://www.open-mpi.org [accessed on April 2011]
31. Philippsen M, Haumacher B, Nester C. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience* 2000; **12**(7):495–518.
32. Kurzyniec D, Wrzosek T, Sunderam V, Slominski A. RMIX: a multiprotocol RMI framework for Java. *Proceedings of the 5th IEEE International Workshop on Java for Parallel and Distributed Computing (JAVAPDC '03)*, Nice, France, 22–26 April 2003; 1–7.
33. Maassen J, van Nieuwpoort RV, Veldema R, Bal HE, 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. 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.
35. Taboada GL, Teijeiro C, Touriño J. High performance Java remote method invocation for parallel computing on clusters. *Proceedings of the 12th IEEE Symposium on Computers and Communications (ISCC '07)*, Aveiro, Portugal, 1–4 July 2007; 233–239.
36. Bornemann M, van Nieuwpoort RV, Kielmann T. MPJ/Ibis: a flexible and efficient message passing platform for Java. *Proceedings of the 12th European PVM/MPI Users' Group Meeting (Euro PVM/MPI '05)*, Sorrento, Italy, 18–21 September 2005; 217–224.
37. Genaud S, Rattanapoka C. P2P-MPI: a peer-to-peer framework for robust execution of message passing parallel programs on grids. *Journal of Grid Computing* 2007; **5**(1):27–42.
38. Carpenter B, Fox G, Ko SH, Lim SB. Object serialization for marshaling data in a Java interface to MPI. *Concurrency: Practice and Experience* 2000; **12**(7):539–553.
39. Taboada GL, Touriño J, Doallo R. F-MPJ: scalable Java message-passing communications on parallel systems. *Journal of Supercomputing* 2011. DOI: 10.1007/s11227-009-0270-0
40. Zhang BY, Yang GW, Zheng WM. Jcluster: an efficient Java parallel environment on a large-scale heterogeneous cluster. *Concurrency and Computation: Practice and Experience* 2006; **18**(12):1541–1557.
41. Al-Jaroodi J, Mohamed N, Jiang H, Swanson D. JOPI: a Java object-passing interface. *Concurrency and Computation: Practice and Experience* 2005; **17**(7–8):775–795.
42. Kaminsky A. Parallel Java: a unified API for shared memory and cluster parallel programming in 100% Java. *Proceedings of the 9th IEEE International Workshop on Java and Components for Parallelism, Distribution and Concurrency (IWJacPDC '07)*, Long Beach, CA, USA, 26–30 March 2007; 1–8.
43. Bouchenak S, Hagimont D, Krakowiak S, Palma ND, Boyer F. Experiences implementing efficient Java thread serialization, mobility and persistence. *Software: Practice and Experience* 2004; **34**(4):355–393.