J Supercomput
DOI 10.1007/s11227-009-0270-0

F-MPJ: scalable Java message-passing communications on parallel systems

Guillermo L. Taboada · Juan Touriño · Ramón Doallo

Abstract This paper presents F-MPJ (Fast MPJ), a scalable and efficient Message-Passing in Java (MPJ) communication middleware for parallel computing. The increasing interest in Java as the programming language of the multi-core era demands scalable performance on hybrid architectures (with both shared and distributed memory spaces). However, current Java communication middleware lacks efficient communication support. F-MPJ boosts this situation by: (1) providing efficient non-blocking communication, which allows communication overlapping and thus scalable performance; (2) taking advantage of shared memory systems and high-performance networks through the use of our high-performance Java sockets implementation (named JFS, Java Fast Sockets); (3) avoiding the use of communication buffers; and (4) optimizing MPJ collective primitives. Thus, F-MPJ significantly improves the scalability of current MPJ implementations. A performance evaluation on an InfiniBand multi-core cluster has shown that F-MPJ communication primitives outperform representative MPJ libraries up to 60 times. Furthermore, the use of F-MPJ in communication-intensive MPJ codes has increased their performance up to seven times.

Keywords Message-Passing in Java (MPJ) · Scalable parallel systems · Communication middleware · Scalable collective communication · High-Performance Computing · Performance evaluation

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

J. Touriño
e-mail: juan@udc.es

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

Published online: 27 February 2009

1 Introduction

Java has become a leading programming language, especially for distributed programming, and is an emerging option for High-Performance Computing (HPC). The increasing interest on Java for parallel computing is based on its appealing characteristics: built-in networking and multithreading support, object orientation, platform independence, portability, security, and it is the main training language for computer science students and has a wide community of developers. Moreover, performance is no longer an obstacle. The gap between Java and native language performance has been narrowing for the last years, thanks to the Just-in-Time (JIT) compiler of the Java Virtual Machine (JVM) that obtains native performance from Java bytecode. Nevertheless, although the performance gap is usually small for sequential applications, it can be particularly high for parallel applications when depending on communications performance. The main reason is the lack of efficient Java communication middleware, which has hindered Java adoption for HPC.

Regarding HPC platforms, new deployments are increasing significantly the number of cores installed in order to meet the ever growing computational power demand. This current trend to multi-core clusters underscores the importance of parallelism and multithreading capabilities [8]. Therefore, this scenario requires scalable parallel solutions, where communication efficiency is fundamental. This efficiency not only depends heavily on the network fabric, but more and more on the communication middleware. Furthermore, hybrid systems (shared/distributed memory architectures) increase the complexity of communication protocols as they have to combine inter-node and intra-node communications, which may imply efficient communication overlapping. Hence, Java represents an attractive choice for the development of communication middleware for these systems as it is a multithreaded language, supports the heterogeneity of the systems and can rely on efficient communication middleware that provides support on high-performance communication hardware. Thus, Java can take full advantage of hybrid architectures using intra-process communication in shared memory and relying on efficient inter-node communication. Moreover, Java can handle the increasing availability of computing resources thanks to its portability and the use of scalable communication middleware. Therefore, as scalability is a key factor to confront new challenges in parallel computing, we aim at providing such feature in Java message-passing middleware through the use of efficient non-blocking communications and high-speed networks support. Furthermore, MPJ collective primitives must implement scalable algorithms. Our F-MPJ (Fast MPJ) library addresses all these issues.

The structure of this paper is as follows: Sect. 2 presents background information and introduces related work. Section 3 describes the design of F-MPJ. The novel issues in its implementation, together with its communication algorithms operation, are shown in Sect. 4. The implementation details on different underlying communication libraries are also covered in this section. Once the basic point-to-point communication methods have been described, the development details of the message-passing collective primitives are presented in Sect. 5. Comprehensive benchmark results from an F-MPJ evaluation on an InfiniBand multi-core cluster are shown in Sect. 6. This evaluation consists of a microbenchmarking of point-to-point and collective primi-

tives, and also a kernel/application benchmarking. Finally, Sect. 7 concludes the paper.

2 Related work

Since the introduction of Java, there have been several implementations of Java messaging libraries for HPC [15]. These libraries have followed different implementation approaches: (1) using Java Remote Method Invocation (RMI), (2) wrapping an underlying native messaging library like MPI [13] through Java Native Interface (JNI), or (3) using low-level Java sockets. Each solution fits with specific situations, but presents associated trade-offs. Using a "pure" Java (100% Java) approach when basing on Java RMI ensures portability, but it might not be the most efficient solution, especially in the presence of high-performance hardware. The use of JNI has portability problems, although usually in exchange for higher performance. The use of a low-level API, Java sockets, requires an important programming effort, especially in order to provide scalable solutions, but it significantly outperforms RMI-based communication libraries.

Although most of the Java communication middleware is based on RMI, MPJ libraries looking for efficient communication have followed the latter two approaches. Thus, mpiJava [1] is a wrapper library that resorts to MPI for communications. However, although its performance is usually high, mpiJava currently only supports some native MPI implementations, as wrapping a wide number of functions and heterogeneous runtime environments entails an important maintenance effort. Additionally, this implementation is not thread-safe, being unable to take advantage of multi-core systems through multithreading. As a result of these drawbacks, the mpiJava maintenance has been superseded by the development of MPJ Express [3], a "pure" Java MPJ library based on mpiJava and implemented on top of the Java New I/O package (Java NIO). MPJ Express is thread-safe and implements a pluggable architecture that combines the portability of the "pure" Java NIO communications with the high-performance Myrinet support (through the native Myrinet eXpress, MX, communication library).

MPJ/Ibis [5] is another MPJ library. It has been implemented on top of Ibis [20], a parallel and distributed Java computing framework. Ibis can use either "pure" Java communications, or native communications on Myrinet. There are two low-level communication devices in Ibis: TCPIbis, based on Java IO sockets (TCP), and NIOIbis, which provides blocking and non-blocking communication through Java NIO sockets. Nevertheless, MPJ/Ibis is not thread-safe, does not take advantage of non-blocking communication, and its Myrinet support is based on the GM library, which shows poorer performance than the MX library.

The two latter libraries, MPJ Express and MPJ/Ibis, are the most active projects in terms of adoption by the HPC community, presence on academia and production environments, and available documentation. These projects are also stable and publicly available along with their source code. Therefore, they have been selected as representative MPJ libraries for the performance evaluation (Sect. 6).

Additionally, there are several recent Java message-passing projects, such as Parallel Java [12], Jcluster [21] and P2P-MPI [10], projects tailored to hybrid, heterogeneous and grid computing systems, respectively. However, their analysis in the performance evaluation section was discarded as a preliminary evaluation of these libraries showed lower scalability than MPJ Express and MPJ/Ibis. Previous Java message-passing libraries, of which eleven projects are cited in [15], although raised many expectations in the past, are currently out-of-date and their interest is quite limited. However, it is worth mentioning MPJava [14] as it was the first Java message-passing library in taking advantage of the scalability and high-performance communications of Java NIO sockets. This important number of past and present projects is a result of the sustained interest in the use of Java for parallel computing.

3 Overview of the F-MPJ communication support

Figure 1 presents an overview of the F-MPJ layered design on representative HPC hardware. From top to bottom, it can be seen that a message-passing application in Java (MPJ application) calls F-MPJ point-to-point and collective primitives. These primitives implement the MPJ communications API on top of the `xxdev` layer, which has been designed as a pluggable architecture and provides a simple but powerful API. This design eases the development of new communication devices in order to provide custom implementations on top of specific native libraries and HPC hardware. Thus, `xxdev` is portable as it presents a single API and provides efficient communication on different system configurations. The use of pluggable low-level communication devices has been already proposed by MPJ Express in its `xdev` communication layer [2]. The `xxdev` (eXtended `xdev`) layer follows the `xdev` approach although adding additional functionality (e.g., allowing the communication of

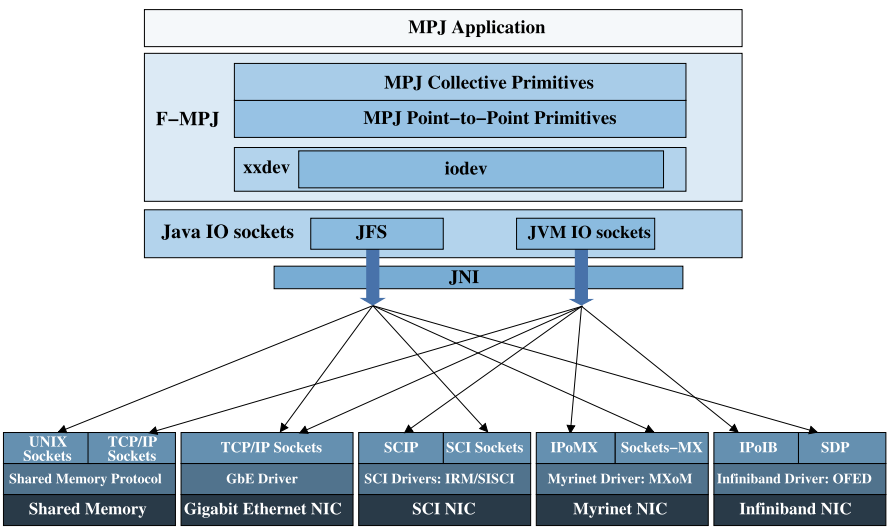

Fig. 1 Overview of F-MPJ communication layers on HPC hardware

any serializable object without data buffering). The motivation of this design decision is to favor the integration of these low-level communication devices in different MPJ libraries, trying to standardize the use of the `xdev`/`xxdev` API in low-level Java communication libraries.

Currently, F-MPJ includes an implementation of `xxdev` using Java IO sockets, `iodev`. This communication device accesses HPC hardware through JNI using either standard JVM IO sockets (TCP) or Java Fast Sockets (JFS) [16], a high-performance Java IO sockets (TCP) implementation, as it can be seen in Fig. 1. For clarity purposes, we denote the sockets IO API as "Java IO sockets." Two implementations of Java IO sockets are considered in this paper: the default JVM IO sockets and JFS.

The HPC systems supported are shared memory machines, high-speed network clusters such as Gigabit Ethernet, SCI, Myrinet and InfiniBand clusters, or hybrid shared/distributed memory systems, such as multi-core high-speed clusters. Figure 1 also shows the different high-performance native libraries that provide communication support over this HPC hardware. On SCI, Myrinet and InfiniBand the available libraries are IP emulations (SCIP, IPoMX and IPoIB) and high-performance native sockets libraries (SCI Sockets, Sockets-MX and Sockets Direct Protocol, SDP) available. IP emulations usually provide wider support but at a higher communication overhead than high-performance native sockets. In fact, JVM IO sockets are usually only supported by IP emulations. The native libraries accessed by JFS and the default JVM IO sockets are presented below the JNI layer. Thus, F-MPJ provides efficient communication over high-performance native libraries through the use of JFS, if available. If JFS is not available, F-MPJ resorts to HPC hardware through the standard JVM IO sockets and IP emulations, maintaining the portability of the solution. Furthermore, F-MPJ relies on these low-level native libraries for lost message detection and error recovery, although F-MPJ communication primitives return proper error codes on communication failures in order to reduce runtime errors. The design and implementation details of the F-MPJ operation are presented in the next section.

4 F-MPJ low-level communication device: *xxdev*

The low-level `xxdev` API provides only basic point-to-point communications. Java IO sockets have been selected for the `xxdev` implementation included in F-MPJ, `iodev`, in order to take advantage of their simple operation and the high-speed networks support of JFS, a Java IO sockets implementation. Thus, `iodev` can rely either on JVM IO sockets or on JFS. This combination of a portable JVM-based implementation with a custom solution for HPC native libraries provides both portability and high performance. Furthermore, Java IO sockets have also been selected as the performance evaluation presented in Sect. 6 has shown that MPJ/Ibis, library based on Java IO sockets, outperforms MPJ Express, implemented on top of Java NIO sockets. Although the better results of MPJ/Ibis could be due to its implementation itself, we have checked that the underlying socket implementation has an important impact on the overall performance. The use of RMI and asynchronous Java sockets [11] has also been discarded due to its high overhead and the lack of portability, respectively. Furthermore, both solutions do not provide high-speed networks support. However,

the use of Java IO sockets requires a significant effort in developing scalable non-blocking communications, features directly provided by Java NIO sockets, but not by Java IO sockets. Next subsections present the `xxdev` API, its communication protocols implementation and its efficient JFS support on HPC native libraries.

4.1 *xxdev* API design

The `xxdev` API has been designed with the goal of being simple, providing only basic communication methods in order to ease the development of `xxdev` devices. A communication device is similar to an MPI communicator, but with reduced functionality. Thus, the `xxdev` API, presented in Listing 1, is composed of 13 methods. Moreover, its API extends the MPJ Express `xdev` API, allowing the communication of any serializable object instead of being limited to transfer only the custom MPJ Express buffer objects. The `newInstance` method instantiates the pluggable `xxdev` device implementations. The `init` method first reads machine names, ports and ranks from a config file (passed as a parameter in `args`), creates the connections, disables Nagle's algorithm and increases socket buffer size (512 KB). Then, the identification of the initialized device is broadcast through all the open connections. Finally, the identifiers of the communication peers are gathered in order to complete the initialization. The `id` method returns the identifier (`id`) of the device. The `finish` method is the last method to be called and completes the device operation.

The `xxdev` communication primitives only include point-to-point communication, both blocking (`send` and `recv`, like MPI_Send and MPI_Recv) and non-blocking (`isend` and `irecv`, like MPI_Isend and MPI_Irecv). Synchronous communications are also embraced (`ssend` and `issend`). These methods use as `dst` (destination) and `src` (source) parameters the ranks read from the config file. The `probe` method waits until a message matching `src`, `tag` and `context` arrives. Its non-blocking version, `iprobe`, checks if the message has been received. The `peek` method (blocking) returns the most recently completed `Request` object, useful for the `Request.iwaitany` implementation. Listing 2 presents the API of the `Request` class, whose wait methods are used to complete the non-blocking communications. Despite the simplicity of the `xxdev` API, the F-MPJ library implements its communications exclusively on top of it, making an intensive use of non-blocking methods for communication overlapping.

4.2 The *iodev* low-level communication device

The `iodev` device implements the low-level multiplexed, non-blocking communication primitives on top of Java IO sockets. In `iodev` each process is connected to every other process through two TCP sockets, one for sending and another for receiving. This is a design decision in order to reduce synchronization overheads when sending/receiving data to/from the same peer process. The access to these sockets, both for reading and writing, is controlled by locks, as several threads have read-/write access to these sockets.

In `iodev` all communication methods are based on the non-blocking primitives `isend`/`irecv`. Thus, blocking communication methods are implemented as a non-blocking primitive followed by an `iwait` call. In order to handle the non-blocking

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

 public Request isend(Object buf, int dst, int tag);
 public Request irecv(Object buf, int src, int tag, Status stts);
 public void send(Object buf, int dst, int tag);
 public Status recv(Object buf, int src, int tag);

 public Request issend(Object buf, int dst, int tag);
 public void ssend(Object buf, int dst, int tag);

 public Status iprobe(int src, int tag, int context);
 public Status probe(int src, int tag, int context);
 public Request peek();
}
```

Listing 1 Public interface of the *xxdev.Device* class

```
public class Request {
 public Status iwait();
 public static Status iwaitany(Request[] reqs)
 public Status test();

 public boolean cancel();
 public void free();
 public boolean isnull();
}
```

Listing 2 Public interface of the `xxdev.Request` class

communications their `Request` objects are internally stored in two sets named `pending_sendRequestSet` and `pending_recvRequestSet`.

An `iodev` message consists of a header plus data. The message header includes the `datatype` sent, the source identification `src`, the message `size`, the `tag`, 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 is 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. When the source of a message is equal to its destination the socket communication is replaced by an array copy.

Regarding message identification, in `iodev` a message is unequivocally identified by the triplet `src`, `tag` and `context`, although the wildcard values `xxdev.Device.ANY_SRC` and `xxdev.Device.ANY_TAG` skip `src` and `tag` matching, respectively. In `iodev` the message reception is carried out by both the `input handler`, a thread in charge of the message reception (also known in the literature as the progress engine), and the `Request.iwait` method. Usually, in message-passing libraries, both native and Java implementations, only the `input`

```
Method Request.iwait():Status
if alreadyCompleted then
    return status;
init_timer();
while completed = false do
    receive_data();    // sets completed ← true if the requested data is received
    if timer_elapsed > max_polling−time then
        current_thread_yield();
        reset_timer();
status ← new Status(statusDetails);
alreadyCompleted ← true;
return status;
```

Fig. 2 *Request.iwait* method pseudocode

`handler` receives messages. This presents a high reception overhead that consists of: (1) the reception of the message by the `input handler`; (2) the notification of the reception to the `Request` object, which is in a `wait` state; (3) waking up the `Request` object; and (4) context switching between the `input handler` and the `Request`, in order to continue the process execution. However, in F-MPJ both the `input handler` thread and the `Request.wait` method receive messages. Thus, if `Request.iwait` receives the message the overhead of the `input handler` reception is avoided.

Figure 2 shows the `Request.iwait` pseudocode in order to illustrate its reception operation. It can be seen that `iodev` implements a polling strategy together with periodically issued yield calls, which decrease `iwait` thread priority in order to not monopolize system CPU. This strategy allows to reduce the message latency significantly in exchange for a moderate CPU overhead increase, compared with the approach where only the `input handler` receives data. This `iodev` approach yields significant benefits, especially in communication-intensive codes, as message latency reduction provides higher scalability than the availability of more CPU power.

4.3 *iodev* communication protocols

The `iodev` device implements the eager and rendezvous protocols, targeted to short and long messages, respectively. The threshold between these protocols is configurable and usually ranges from 128 to 512 KB.

4.3.1 iodev eager protocol

The eager protocol is targeted to short messages, typically below 128 KB. It is based on the assumption that the receiver has available storage space, so there is no exchange of control messages before the actual data transfer. This strategy minimizes the overhead of control messages, that can be significant for short messages.

Figure 3 shows eager protocol pseudocode. Regarding eager `isend` operation, the sender writes the data under the assumption that the receiver will handle it. At the receiver side there are two possible scenarios for the `input handler` (see

Method `isend(`*buffer,dst,tag,context*`)`:Request (Eager)

```
sendRequest ← new SendRequest(buffer,dst,tag,context);
send(dst,buffer);
sendRequest.completed ← true;
return sendRequest;
```

Method `input handler thread` (Eager)

```
while running do
    receive_header(messageHeader);
    rRequest ← new RecvRequest(messageHeader);
    if rRequest in pending_recvRequestSet then
        recvRequest ← pending_recvRequestSet.remove(rRequest);
        recvRequest.buffer ← receive_data();
    else
        rRequest.temp_buffer ← receive_data();
        pending_recvRequestSet.add(rRequest);
```

Method `irecv(`*buffer,src,tag,context,status*`)`:Request (Eager)

```
rRequest ← new RecvRequest(buffer,src,tag,context,status);
if rRequest in pending_recvRequestSet then
    recvRequest ← pending_recvRequestSet.remove(rRequest);
    buffer ← recvRequest.temp_buffer;
    return recvRequest;
else
    pending_recvRequestSet.add(rRequest);
    return rRequest;
```

Fig. 3 *iodev eager* protocol pseudocode

pseudocode in Fig. 3), depending on whether a matching receive has been already posted or not. Thus, if a matching `recvRequest` exists the message is copied into the destination buffer; otherwise, it will be stored in a temporary buffer, waiting for the corresponding `irecv` post. The `input handler` is constantly running during `iodev` operation, from the `init` to the `finish` call. This behavior is controlled by a flag (`running`). The `irecv` operation (see Fig. 3) also presents two scenarios, depending on whether the input handler has already received the message or not. This `iodev` eager protocol implementation reduces significantly F-MPJ short message overhead, allowing short message communication-intensive MPJ applications to increase significantly their scalability.

4.3.2 iodev rendezvous protocol

The rendezvous protocol is targeted to long messages, typically above 128 KB. It is based on the use of control messages in order to avoid buffering. Thus, the steps of the protocol are: (1) the source sends a ready-to-send message; (2) the destination replies with a ready-to-receive message; and (3) data is actually transferred. This strategy avoids buffering although increases protocol overhead. However, the impact of the control messages overhead is usually small for long messages.

```
Method isend(buffer,dst,tag,context):Request (Rendezvous)
  sendRequest ← new SendRequest(buffer,dst,tag,context);
  pending_sendRequestSet.add(sendRequest);
  send(dst,ready-to-send_Message);
  sendRequest.completed ← false;
  return sendRequest;

Method input handler (Rendezvous)
  while running do
      messageHeader ← receive_header();
      request ← new Request(messageHeader);
      if messageHeader from a ready-to-send_Message then
          if request in pending_recvRequestSet then
              pending_recvRequestSet.remove(request);
              send(src,ready-to-recv_Message);
          else
              pending_recvRequestSet.add(request);
      else if messageHeader from a ready-to-receive_Message then
          Fork: rendez_Write_Thread:
          begin
              sendRequest ← pending_sendRequestSet.remove(request);
              send(dst,sendRequest.buffer);
              sendRequest.completed ← true;
          end
      else if messageHeader from a dataMessage then
          recvRequest ← pending_recvRequestSet.remove(request);
          recvRequest.buffer ← receive_data();

Method irecv(buffer,src,tag,context,status):Request (Rendezvous)
  rRequest ← new RecvRequest(buffer,src,tag,context,status);
  if rRequest in pending_recvRequestSet then
   | send(src,ready-to-recv_Message);
  else
   |pending_recvRequestSet.add(rRequest);
  return rRequest;
```

Fig. 4 *iodev rendezvous* protocol pseudocode

Figure 4 shows rendezvous protocol pseudocode. The `isend` operation consists of writing a ready-to-send control message. At the receiver side there are three possible scenarios for the `input handler` (see pseudocode in Fig. 4), depending on the incoming message: (1) a ready-to-send message; (2) a ready-to-receive message; or (3) a data message. In scenario (1) a ready-to-receive message reply is written if a matching receive has been posted; otherwise, the ready-to-send message is stored until such matching receive is posted. In (2) the actual transfer of the data is performed through a forked thread in order to avoid `input handler` blockade while writing data. In this case the `input handler` is run by the sender process and therefore can access the source buffer. Finally, in (3) the `input handler` receives the data. The `irecv` operation (see Fig. 4) presents two scenarios, depending on whether the input handler has already received the ready-to-send message or not. Thus, it either replies back with a ready-to-receive message or stores the receive post, respectively. This `iodev` rendezvous protocol implementation contributes significantly to F-MPJ scalability as it prevents from message buffering and network congestion. Therefore, scalable Java communication performance can be achieved.

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

Listing 3 JFS extended API for communicating primitive data type arrays directly

```
if (os instanceof jfs.net.SocketOutputStream) {
  jfsAvailable = true;
  jfsos = (jfs.net.SocketOutputStream) os;
}
oos = new ObjectOutputStream(os);

[...]

// Writing int_array[offset] ... int_array[offset+num_elements-1]
if (jfsAvailable)
  jfsos.write(int_array, offset, num_elements);
else {
  int[] intBuf = (int[]) Array.newInstance(int.class, num_elements);
  System.arraycopy(int_array, offset, intBuf, 0, num_elements);
  oos.writeUnshared(intBuf);
}
```

Listing 4 JFS direct send of part of an int array

4.4 Java Fast Sockets support in *iodev*

The default sockets library used by `iodev`, JVM IO sockets, presents several disadvantages for communication middleware: (1) this library has to resort to serialization, the process of transforming objects (except byte arrays) in byte series, for message communication; (2) as Java cannot serialize/deserialize array portions (except for parts of byte arrays) a new array must be created to store the portion to be serialized/deserialized; (3) JVM IO sockets perform an extra copy between the data in the JVM heap and native memory in order to transfer the data; and finally, (4) this socket library is usually not supported by high-performance native communication libraries, so it has to rely on IP emulations, a solution which presents a poorer performance.

However, in order to avoid these drawbacks, F-MPJ has integrated in `iodev` the high-performance Java sockets library JFS (Java Fast Sockets) [16], in a portable and efficient way. Thus, JFS boosts F-MPJ communication efficiency by: (1) avoiding primitive data type array serialization through an extended API that allows direct communication of primitive data type arrays (see Listing 3); (2) making unnecessary the data buffering when sending/receiving portions of primitive data type arrays using `offset` and `length` parameters (see JFS API in Listing 3 and its application in Listing 4); (3) avoiding the copies between the JVM data and native memory thanks to JFS's zero-copy protocol; and (4) providing efficient support on shared memory, and Gigabit Ethernet, SCI, Myrinet and InfiniBand networks through the use of the underlying libraries specified in Fig. 1.

Listing 4 presents an example of `iodev` code that takes advantage of the efficient JFS methods when they are available, without compromising the portability of the

solution. This handling of JFS communications is of special interest in F-MPJ and, in general, in any communication middleware, as MPJ applications can benefit from the use of JFS without modifying their source code. The integration of JFS in `iodev` has been done following this approach and thus preserving F-MPJ portability while taking full advantage of the underlying communication middleware. In fact, JFS, in the presence of two or more supported libraries, prioritizes them depending on their performance: usually shared memory communication first, then high-performance socket libraries, and finally the default "pure" Java implementation.

JFS significantly outperforms JVM IO sockets, especially in shared memory and hybrid shared/distributed memory architectures. Moreover, JFS is targeted to primitive data type array communications, frequently used in HPC applications. Therefore, F-MPJ benefits especially from the use of JFS, as will be experimentally assessed in Sect. 6.

5 Implementation of Java message-passing collective primitives

As `iodev` already provides the basic point-to-point primitives, their implementation in F-MPJ is direct. Nevertheless, collective primitives require the development of algorithms that involve multiple point-to-point communications. MPJ application developers use collective primitives for performing standard data movements (e.g., broadcast, scatter and gather) and basic computations among several processes (reductions). This greatly simplifies code development, enhancing programmers productivity together with MPJ programmability. Moreover, it relieves developers from communication optimization. Thus, collective algorithms must provide scalable performance, usually through overlapping communications in order to maximize the number of operations carried out in parallel. An unscalable algorithm can easily waste the performance provided by an efficient communication middleware.

5.1 MPJ collective algorithms

The design, implementation and runtime selection of efficient collective communication operations have been extensively discussed in the context of native message-passing libraries [4, 7, 17, 19], but not in MPJ. Therefore, F-MPJ has tried to adapt the research in native libraries to MPJ. As far as we know, this is the first project in this sense, as up to now MPJ library developments have been focused on providing production quality implementations of the full MPJ specification, rather than scalable performance for collective implementations.

The collective algorithms present in MPJ libraries can be classified in five types, namely Flat Tree (FT) or linear, Minimum-Spanning Tree (MST), Binomial Tree (BT), Bucket (BKT) or cyclic, and BiDirectional Exchange (BDE) or recursive doubling.

The simplest algorithm is FT, where all communications are performed sequentially. Figure 5 shows the pseudocode of the FT broadcast using either blocking primitives (from now on denoted as bFT) or exploiting non-blocking communications (from now on nbFT) in order to overlap communications. As a general rule,

Method `Bcast(x,root)(bFT)`

```
if me = root then
    for i=0,...,npes-1 do
        if me != i then
            SEND (x,p_i);
else
    RECV (x,root);
```

Method `Bcast(x,root)(nbFT)`

```
if me = root then
    for i=0,...,npes-1 do
        if me != i then
            sreq_i = ISEND (x,p_i);
else
    rreq = IRECV (x,root);
    WAIT (rreq);

if me = root then
    for i=0,...,npes-1 do
        if me != i then
            WAIT (sreq_i);
```

Fig. 5 FT broadcast pseudocode

Method `MSTBcast(x,root,left,right)`

```
if left = right then return;
mid = ⌊(left+right)/2⌋;
if root ≤ mid then dest=right; else dest = left;
if me = root then SEND (x,dest);
if me = dest then RECV (x,root);
if me ≤ mid and root ≤ mid then MSTBCAST (x,root,left,mid);
else if me ≤ mid and root > mid then MSTBCAST (x,dest,left,mid);
else if me > mid and root ≤ mid then MSTBCAST (x,dest,mid+1,right);
else if me > mid and root > mid then MSTBCAST (x,root,mid+1,right);
```

Fig. 6 MSTBcast pseudocode

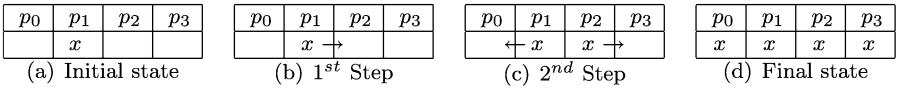

Fig. 7 Minimum-spanning tree algorithm for broadcast

valid for all collective algorithms, the use of non-blocking primitives avoids unnecessary waits and thus increases the scalability of the collective primitive. However, for the FT broadcast only the send operation can be overlapped. The variables used in the pseudocode are also present in the following figures. Thus, `x` is the message, `root` is the root process, `me` is the rank of each parallel process, $\mathtt{p}_i$ the i-th process and `npes` is the number of processes used.

Figures 6 and 7 present MST pseudocode and operation for the broadcast, which is initially invoked through `MSTBcast(x,root,0,npes-1)`. The parameters `left` and `right` indicate the indices of the left- and right-most processes in the current subtree. A variant of MST is BT, where for each step i (from 1 up to $\lceil \log_2(npes) \rceil$) the process $\mathtt{p}_j$ communicates with the process $\mathtt{p}_{j+2^{i-1}}$.

Method BKTAllgather(x)

```
prev = me - 1;
if prev < 0 then prev= npes - 1;
next = me + 1;
if next = npes then next=0;

current_i = me;
for i=0,...,npes-2 do
    current_i = current_i - 1;
    if current_i < 0 then current_i=npes-1;
    rreq_i=IRECV (x_current_i,prev);
BARRIER ();
current_i = me;
for i=0,...,npes-2 do
    sreq_i=ISEND (x_current_i,next);
    current_i = current_i - 1;
    if current_i < 0 then current_i=npes-1;

for i=0,...,npes-2 do
    WAIT (sreq_i);
    WAIT (rreq_i);
```

Method BDEAllgather(x,*left*,*right*)

```
if left = right then return;
size = right − left + 1;
mid = ⌊(left+right)/2⌋;

if me ≤ mid then
    partner = me + ⌊size/2⌋;
else
    partner = me − ⌊size/2⌋;

if me ≤ mid then
    BDEALLGATHER (x,left,mid);
else
    BDEALLGATHER (x,mid+1,right);

if me ≤ mid then
    SEND (x_left:mid,partner);
    RECV (x_mid+1:right,partner);
else
    SEND (x_mid+1:right,partner);
    RECV (x_left:mid,partner);
```

Fig. 8 BKTAllgather and BDEAllgather pseudocode

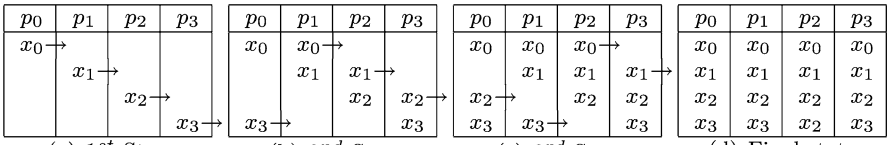

Fig. 9 Bucket algorithm for allgather (BKTAllgather)

Figures 8 (left) and 9 show BKT allgather pseudocode and operation. In BKT all processes are organized like a ring and send at each step data to the process at their right. Thus, data eventually arrives to all nodes. F-MPJ implements an optimization posting all `irecv` requests at BKT start-up. A subsequent synchronization (barrier) prevents early communication that incurs in buffering overhead when the `irecv` has not already been posted. The communications overlapping is achieved through `isend` calls. Finally, the algorithm waits for the completion of all requests. Figures 8 (right) and 10 present BDE allgather pseudocode and operation, which requires that `npes` be a power of two. In BDE the message size exchanged by each process pair is recursively doubled at each step until data eventually arrives to all nodes.

Although there is a wide variety of collective algorithms, current MPJ libraries mainly resort to FT implementations. Moreover, it is usually provided only one implementation per primitive. Nevertheless, F-MPJ is able to use up to three algorithms per primitive, selected at runtime. Next subsection presents the details of F-MPJ collectives implementation and a comparative analysis of algorithms present in MPJ collective primitives.

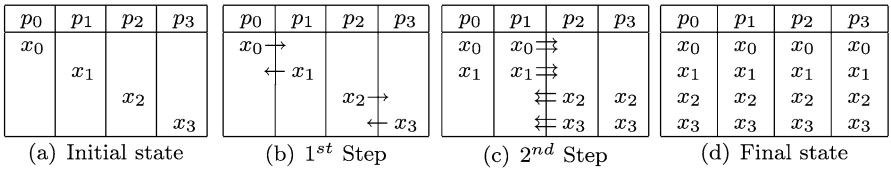

Fig. 10 Bidirectional exchange algorithm for allgather (BDEAllgather). In the 2nd step, bidirectional exchanges occur between the two pairs of processes p_0 and p_2, and p_1 and p_3

5.2 Analysis of the implementation of MPJ collective primitives

Table 1 presents a complete list of the collective algorithms used in F-MPJ, MPJ Express and MPJ/Ibis. It can be seen that F-MPJ implements algorithms with usually higher scalability than MPJ Express and MPJ/Ibis collective primitives, taking advantage of communications overlapping. Thus, MPJ/Ibis only uses non-blocking communications in alltoall and alltoallv primitives, and MPJ Express resorts to bFT, an algorithm with poor scalability, for broadcast (Bcast) and reduce. However, MPJ Express limits in the broadcast implementation the maximum number of transfers per process to four, making up a four-ary tree, in order to alleviate the communication overhead of the root process in bFT.

F-MPJ implements up to three algorithms per primitive. The algorithm selection depends on the message size, using the algorithms with the lowest latencies for short message communication and minimizing message buffering for long message communication. Table 1 indicates the selected algorithms using superscripts. The message size threshold used in this selection is configurable (32 KB by default) and independent for each primitive. The use of efficient communications and scalable algorithms in F-MPJ provides scalable MPJ performance, as will be assessed in the next section.

6 Performance evaluation

6.1 Experimental configuration

F-MPJ has been evaluated on the Finis Terrae supercomputer [9], ranked #209 in June 2008 TOP500 list [18] (14 TFlops). This system is an InfiniBand multi-core cluster that consists of 142 HP Integrity rx7640 nodes, each of them with 8 Montvale Itanium2 (IA64) dual-core processors at 1.6 GHz and 128 GB of memory. The InfiniBand NIC is a dual 4X IB port (16 Gbps of theoretical effective bandwidth). The native low-level communication middleware is SDP and Open Fabrics Enterprise Distribution (OFED) 1.2 (see Sect. 3 and Fig. 1 for further details). The OS is SUSE Linux Enterprise Server 10. The evaluated MPJ libraries are F-MPJ with JFS 0.3.1, MPJ Express 0.27 and MPJ/Ibis 1.4. The JVM is BEA JRockit 5.0 (R27.5), the JVM 1.5 or higher (prerequisite for the evaluated MPJ libraries) that achieves the best performance on Linux IA64.

The evaluation presented in this section consists of a microbenchmarking of point-to-point primitives (Sect. 6.2) and collective communications (Sect. 6.3); and a kernel/application benchmarking of codes from the Java Grande Forum (JGF) Benchmark Suite [6] (Sect. 6.4).

Table 1 Collective algorithms used in representative MPJ libraries

Collective	F-MPJ	MPJ Express	MPJ/Ibis
Barrier	MST	nbFTGather+ bFTBcast	bFT
Bcast	MST[a]	bFT	BT
	MSTScatter+BKTAllgather[b]		
Scatter	MST[a]	nbFT	bFT
	nbFT[b]		
Scatterv	MST[a]	nbFT	bFT
	nbFT[b]		
Gather	MST[a]	nbFT	bFT
	nbFT[b]		
Gatherv	MST[a]	nbFT	bFT
	nbFT[b]		
Allgather	MSTGather+MSTBcast[a]	nbFT	BKT (double ring)
	BKT[b]		
	BDE[c]		
Allgatherv	MSTGatherv+MSTBcast	nbFT	BKT
Alltoall	nbFT	nbFT	nbFT
Alltoallv	nbFT	nbFT	nbFT
Reduce	MST[a]	bFT	BT (commutative)
	BKTReduce_scatter+		bFT (non commu-
	MSTGather[b]		tative operation)
Allreduce	MSTReduce+MSTBcast[a]	BT	BDE
	BKTReduce_scatter+		
	BKTAllgather[b]		
	BDE[c]		
Reduce_scatter	MSTReduce+MSTScatterv[a]	bFTReduce +	{BTReduce *or*
	BKT[b]	nbFTScatterv	bFTReduce}+
	BDE[c]		bFTScatterv
Scan	nbFT	nbFT	bFT

[a]Selected algorithm for short messages

[b]Selected algorithm for long messages

[c]Selectable algorithm for long messages and *npes* power of two

6.2 Microbenchmarking of MPJ point-to-point primitives

In order to microbenchmark F-MPJ primitives performance it has been used our own microbenchmark suite [15]. Thus, the results shown are the half of the round trip time of a ping-pong test (point-to-point latency) or its corresponding bandwidth. Figure 11 shows point-to-point latencies and bandwidths for MPJ byte and double arrays, data structures frequently used in parallel applications, for intra-node (shared memory) and inter-node (InfiniBand) communication. Moreover, the low-level native com-

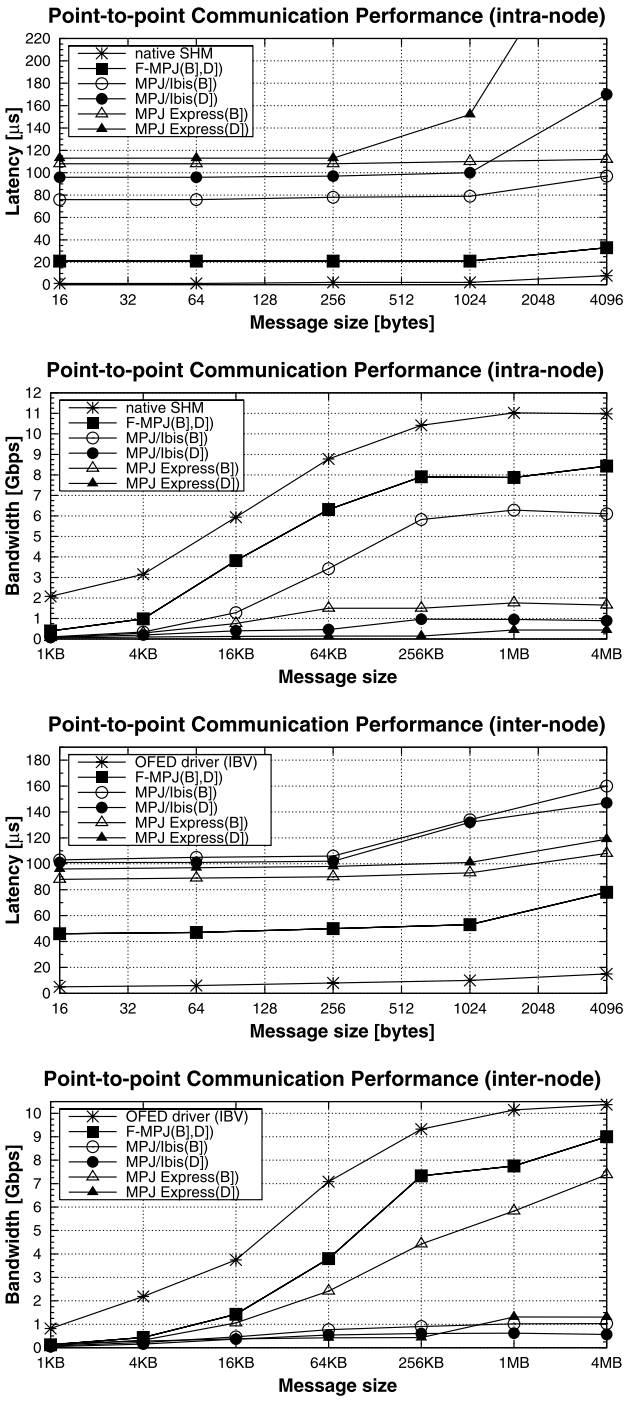

Fig. 11 MPJ point-to-point primitives performance

munication performance of the shared memory protocol and the OFED InfiniBand driver is also shown for comparison purposes. The latency graphs serve to compare short message-performance, whereas the bandwidth graphs are useful to compare long message-performance. For clarity purposes, it has been used the JNI array notation in order to denote byte and double arrays (B] and D], respectively).

F-MPJ, MPJ Express and MPJ/Ibis rely on different sockets implementations (JFS, Java NIO sockets and Java IO sockets, respectively), and thus it is not possible to compare directly the MPJ library processing overhead. However, as the sockets implementations share the same underlying layers, a fair comparison involves the analysis of the overhead of F-MPJ + JFS, MPJ Express + Java NIO and MPJ/Ibis + Java IO sockets. The processing overhead of the MPJ libraries plus socket implementations can be estimated from Fig. 11, where F-MPJ + JFS shows significantly lower overhead than MPJ Express + Java NIO and MPJ/Ibis + Java IO sockets, especially for short messages and double arrays (D]) communication.

F-MPJ handles D] transfers without serialization, obtaining the same results for B] and D] communication. As MPJ/Ibis and MPJ Express have to serialize double arrays, they present a significant performance penalty for D], especially for long messages. Thus, F-MPJ(D]) clearly outperforms MPJ/Ibis(D]) and MPJ Express(D]), showing up to 10 and 20 times higher performance, respectively. The impact of serialization overhead, the difference between D] and B] performance, is especially significant when the MPJ library obtains high B] bandwidths (MPJ/Ibis on intra-node and MPJ Express on inter-node). In these scenarios the serialization is the main performance bottleneck.

The byte array (B]) results are useful for evaluating the data transfer performance itself, without serialization overheads. In this scenario F-MPJ significantly outperforms MPJ Express and MPJ/Ibis, especially for short messages, thanks to its lower start-up latency. Regarding long message intra-node performance, F-MPJ outperforms MPJ/Ibis up to 40% and MPJ Express up to 5 times. Nevertheless, the results vary for inter-node transfers, where F-MPJ outperforms MPJ/Ibis up to 9 times and MPJ Express up to 70%. In these results the impact of the underlying communication middleware is significant. Thus, the high-performance SDP library only supports F-MPJ and MPJ Express, which obtain significantly higher inter-node performance than MPJ/Ibis, only supported by the low performance IP emulation on InfiniBand (IPoIB).

The observed point-to-point communication efficiency of F-MPJ significantly improves MPJ collective primitives performance, as shown next.

6.3 Microbenchmarking of MPJ collective primitives

It has been evaluated the performance scalability of representative F-MPJ, MPJ/Ibis and MPJ Express collective primitives. Figure 12 presents the aggregated bandwidth for broadcast and sum reduction operations, both for short (1 KB) and long (1 MB) double array messages. The aggregated bandwidth metric has been selected as it takes into account the global amount of data transferred, $message\_size * (npes - 1)$ for both collectives. The broadcast and reduce primitives have been selected as representative data movement and computational primitives, respectively. Finally, the two message

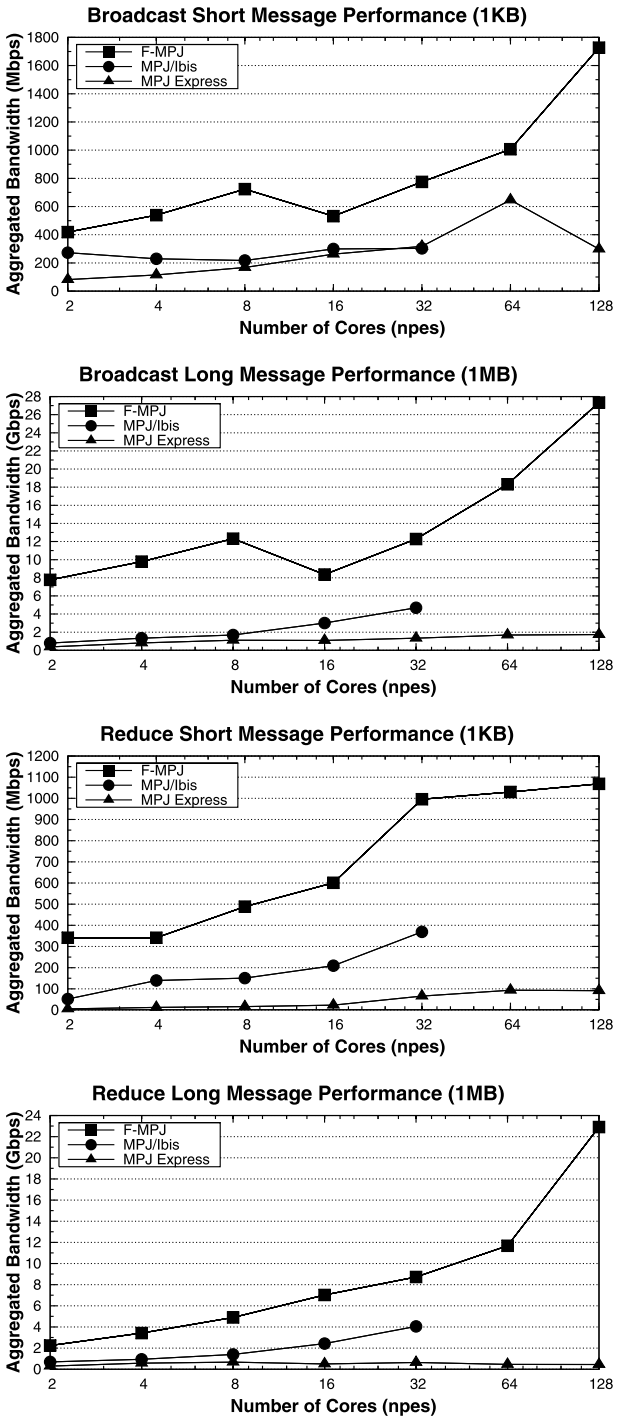

Fig. 12 MPJ collective primitives performance

sizes used are representative of short and long messages. The results have been obtained with a maximum of 8 cores per node as this configuration has shown the best performance. Thus, from now on the number of nodes used is $\lceil npes/8 \rceil$. MPJ/Ibis could not be run in our testbed using more than 32 cores due to an Ibis runtime initialization error.

The presented results (Fig. 12) show that F-MPJ significantly outperforms MPJ Express and MPJ/Ibis. Regarding broadcast, F-MPJ increases performance up to 5.8 and 16 times for short and long messages, respectively. The improvement of the F-MPJ reduce is up to 60 and 50 times for short and long messages, respectively. F-MPJ shows scalable performance for both collectives, obtaining usually the highest performance increases on 128 cores. The higher long message performance improvement of F-MPJ is mainly due to the serialization avoidance. Moreover, F-MPJ takes significant advantage of intra-node communication (up to 8 cores), especially for the broadcast. In fact, F-MPJ broadcast results are better with 8 cores (one node) than with 16 cores (two nodes), where the primitive operation involves the use of inter-node transfers. The lowest performance, especially for the reduce, has been obtained by MPJ Express, whereas MPJ/Ibis results are between F-MPJ and MPJ Express results, although closer to the latter. F-MPJ significantly improves MPJ collectives performance due to the efficient intra-node and inter-node point-to-point communication, the serialization avoidance and the use of scalable algorithms (see Table 1) based on non-blocking communications overlapping.

6.4 MPJ kernel/application performance analysis

The impact of the use of F-MPJ on representative MPJ kernels and applications is analyzed in this subsection. Two kernels and one application from the JGF Benchmark Suite have been evaluated: Crypt, an encryption and decryption kernel; LUFact, an LU factorization kernel; and MolDyn, a molecular dynamics N-body parallel simulation application. These MPJ codes have been selected as they show very poor scalability with MPJ Express and MPJ/Ibis, despite being load balanced parallel implementations. Hence, these are target codes for the evaluation of F-MPJ performance and scalability improvement.

Figure 13 presents Crypt and LUFact speedups. Regarding Crypt, F-MPJ clearly outperforms MPJ/Ibis and MPJ Express, up to 330%, in a scenario where the data transfers (byte arrays) do not involve serialization. Thus, both MPJ/Ibis and MPJ Express take advantage of the use of up to 32 cores. LUFact performs double and integer arrays broadcasts for each iteration of the factorization method. Therefore, the serialization overhead is important for this code. Thus, the use of F-MPJ has a higher impact on performance improvement than for Crypt. Figure 13 (down) shows that F-MPJ significantly outperforms MPJ/Ibis and MPJ Express for LUFact, up to eight times. This performance increase is due to the use of scalable algorithms and the serialization avoidance. Furthermore, F-MPJ presents its best results on 128 cores, whereas MPJ/Ibis and MPJ Express obtain their best performance on 16 and 8 cores, respectively.

The MolDyn application consists of six allreduce sum operations for each iteration of the simulation. The transferred data are integer and double arrays, so F-MPJ can

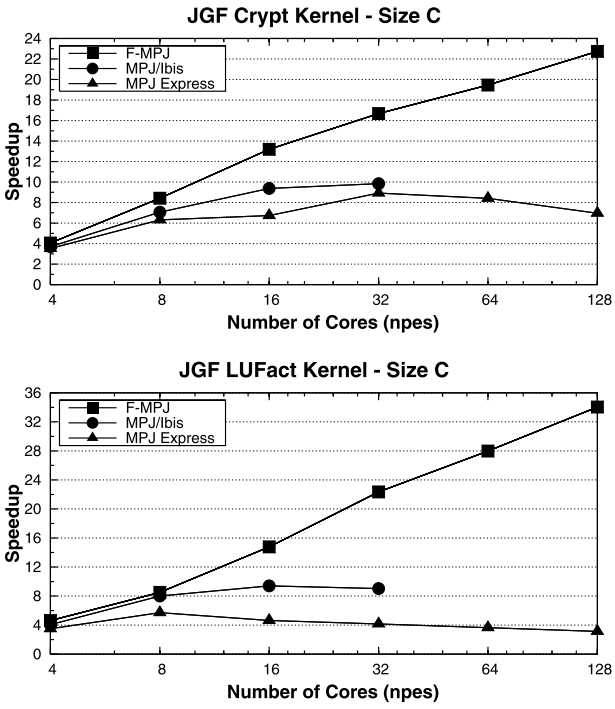

Fig. 13 Speedups of Crypt and LUFact JGF kernels

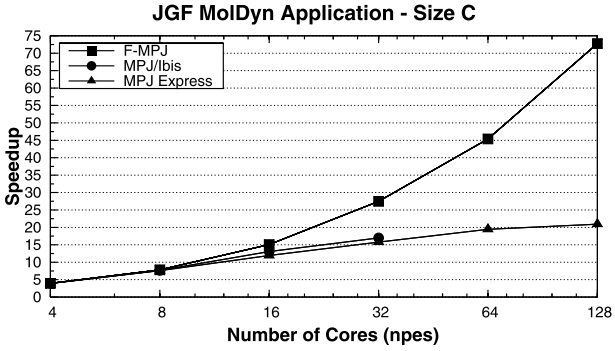

Fig. 14 Speedups of MolDyn JGF application

avoid serialization overhead. For its evaluation it has been used an enlarged size C version ($18 \times 18 \times 18 \times 4$ elements). Figure 14 presents MPJ speedups, where F-MPJ outperforms MPJ/Ibis and MPJ Express up to 3.5 times. This application presents higher speedups than the kernels of Fig. 13 as it is a less communication intensive code, and the three libraries use scalable allreduce algorithms (see Table 1). However,

the serialization overhead negatively affects MPJ/Ibis and MPJ Express MolDyn performance.

The use of F-MPJ increases significantly MPJ kernels and applications performance, especially for communication-intensive codes. Moreover, the scalable F-MPJ performance allows MPJ codes to take advantage of the use of a large number of cores (up to 128 in our experiments), a significantly higher value than that of MPJ/Ibis and MPJ Express.

7 Conclusions

This paper has presented F-MPJ, a scalable and efficient Java message-passing library. The increasing interest in Java parallel solutions on multi-core systems demands efficient communication middleware. F-MPJ pursues to satisfy this need obtaining scalable Java performance in parallel systems. The main contributions of this communication middleware are: (1) provides efficient MPJ non-blocking communication based on Java IO sockets, allowing communications overlapping; (2) it is efficiently coupled with JFS, our high-performance Java IO sockets implementation, which provides shared memory and high-speed networks support and avoids the primitive data type array serialization. (3) F-MPJ avoids the use of communication buffers; and (4) implements scalable Java message-passing collective primitives. F-MPJ has been evaluated on an InfiniBand multi-core cluster, outperforming significantly two representative MPJ libraries, MPJ Express and MPJ/Ibis. Thus, the microbenchmarking results showed a performance increase of up to 60 times for F-MPJ. Moreover, the subsequent kernels and application benchmarking obtained speedup increases of up to seven times for F-MPJ on 128 cores, depending on the communication intensiveness of the analyzed MPJ benchmarks. F-MPJ has improved MPJ performance scalability, allowing Java message-passing codes that previously increased their speedups only up to 8–16 cores to take advantage of the use of 128 cores, improving significantly the performance and scalability of current MPJ libraries.

Further information, additional documentation and software downloads of this project are available from the F-MPJ and JFS Projects webpage http://jfs.des.udc.es.

Acknowledgements This work was funded by the Ministry of Education and Science of Spain under Projects TIN2004-07797-C02 and TIN2007-67537-C03-2 and by the Galician Government (Xunta de Galicia) under Project PGIDIT06PXIB105228PR. We gratefully thank CESGA (Galician Supercomputing Center, Santiago de Compostela, Spain) for providing access to the Finis Terrae supercomputer.

References

1. Baker M, Carpenter B, Fox G, Ko S, Lim S (1999) mpiJava: an object-oriented java: interface to MPI. In: 1st Intl workshop on Java for parallel and distributed computing (IWJPDC'99), San Juan, Puerto Rico, 1999, pp 748–762. http://www.hpjava.org/mpiJava.html (Last visited: December 2008)
2. Baker M, Carpenter B, Shafi A (2005) A pluggable architecture for high-performance Java messaging. IEEE Distrib Syst Online 6(10):1–4
3. Baker M, Carpenter B, Shafi A (2008) MPJ Express: towards thread safe Java HPC. In: Proc 8th IEEE intl conf on cluster computing (Cluster'06), Barcelona, Spain, 2006, pp 1–10

4. Barchet-Estefanel LA, Mounie G (2004) Fast tuning of intra-cluster collective communications. In: Proc 11th European PVM/MPI users' group meeting (EuroPVM/MPI'04), Budapest, Hungary, 2004, pp 28–35
5. Bornemann M, van Nieuwpoort RV, Kielmann T (2005) MPJ/Ibis: a flexible and efficient message passing platform for Java. In: Proc 12th European PVM/MPI users' group meeting (EuroPVM/MPI'05), Sorrento, Italy, 2005, pp 217–224
6. Bull JM, Smith LA, Westhead MD, Henty DS, Davey RA (2000) A benchmark suite for high performance Java. Concurr Pract Exp 12(6):375–388
7. Chan E, Heimlich M, Purkayastha A, van de Geijn RA (2007) Collective communication: theory, practice, and experience. Concurr Comput Pract Exp 19(13):1749–1783
8. Dongarra JJ, Gannon D, Fox G, Kennedy K (2007) The impact of multicore on computational science software. CTWatch Q 3(1):1–10
9. Finis Terrae Supercomputer (2008) http://www.top500.org/system/9156 (Last visited: December 2008)
10. Genaud S, Rattanapoka C (2005) A peer-to-peer framework for robust execution of message passing parallel programs. In: Proc 12th European PVM/MPI users' group meeting (EuroPVM/MPI'05), Sorrento, Italy, 2005, pp 276–284
11. IBM: Asynchronous IO for Java (2008) http://www.alphaworks.ibm.com/tech/aio4j (Last visited: December 2008)
12. Kaminsky A (2007) Parallel Java: a unified api for shared memory and cluster parallel programming in 100% Java. In: Proc 9th intl workshop on Java and components for parallelism, distribution and concurrency (IWJPDC'07), Long Beach, CA, 2007, p 196a (8 pages)
13. Message Passing Interface Forum (2008) http://www.mpi-forum.org (Last visited: December 2008)
14. Pugh B, Spacco J (2003) MPJava: high-performance message passing in Java using Java.nio. In: Proc 16th intl workshop on languages and compilers for parallel computing (LCPC'03), College Station, TX, 2003, pp 323–339
15. Taboada GL, Touriño J, Doallo R (2003) Performance analysis of Java message-passing libraries on fast ethernet, myrinet and SCI clusters. In: Proc 5th IEEE intl conf on cluster computing (Cluster'03), Hong Kong, China, 2003, pp 118–126
16. Taboada GL, Touriño J, Doallo R (2008) Java Fast Sockets: enabling high-speed Java communications on high-performance clusters. Comput Commun 31(17):4049–4059
17. Thakur R, Rabenseifner R, Gropp W (2005) Optimization of collective communication operations in MPICH. Int J High Perform Comput Appl 19(1):49–66
18. TOP500 Supercomputing Sites (2008) http://www.top500.org (Last visited: December 2008)
19. Vadhiyar SS, Fagg GE, Dongarra JJ (2004) Towards an accurate model for collective communications. Int J High Perform Comput Appl 18(1):159–167
20. van Nieuwpoort RV, Maassen J, Wrzesinska G, Hofman R, Jacobs C, Kielmann T, Bal HE (2005) Ibis: a flexible and efficient Java-based grid programming environment. Concurr Comput Pract Exp 17(7–8):1079–1107
21. Zhang BY, Yang GW, Zheng WM (2006) Jcluster: an efficient Java parallel environment on a large-scale heterogeneous cluster. Concurr Comput Pract Exp 18(12):1541–1557

Guillermo L. Taboada received the B.Sc. and M.Sc. degrees in Computer Science from the University of A Coruña, Spain, in 2002. He is currently a Teaching Assistant and a Ph.D. candidate in the Department of Electronics and Systems of the University of A Coruña. His Ph.D. thesis is devoted to the design of efficient Java communications for high-performance computing.

Juan Touriño received the B.Sc. (1993), M.Sc. (1993) and Ph.D. (1998) degrees in Computer Science from the University of A Coruña, Spain. He is currently a Full Professor in the Department of Electronics and Systems at the University of A Coruña. His primary research interest is in the area of high-performance computing, covering a wide range of topics such as architectures, operating systems, networks, compilers, programming languages/libraries, algorithms and applications. Dr. Touriño is coauthor of more than 100 technical papers on these topics.

Ramón Doallo received his B.Sc. and M.Sc. degrees in Physics from the University of Santiago de Compostela, Spain, in 1987, and his Ph.D. in Physics from the same University in 1992. In 1990 he joined the Department of Electronics and Systems of 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.