J Supercomput
DOI 10.1007/s11227-010-0464-5

Design of efficient Java message-passing collectives on multi-core clusters

Guillermo L. Taboada · Sabela Ramos · Juan Touriño · Ramón Doallo

Abstract This paper presents a scalable and efficient Message-Passing in Java (MPJ) collective communication library for parallel computing on multi-core architectures. The continuous increase in the number of cores per processor underscores the need for scalable parallel solutions. Moreover, current system deployments are usually multi-core clusters, a hybrid shared/distributed memory architecture which increases the complexity of communication protocols. Here, Java represents an attractive choice for the development of communication middleware for these systems, as it provides built-in networking and multithreading support. As the gap between Java and compiled languages performance has been narrowing for the last years, Java is an emerging option for High Performance Computing (HPC).

Our MPJ collective communication library increases Java HPC applications performance on multi-core clusters: (1) providing multi-core aware collective primitives; (2) implementing several algorithms (up to six) per collective operation, whereas publicly available MPJ libraries are usually restricted to one algorithm; (3) analyzing the efficiency of thread-based collective operations; (4) selecting at runtime the most efficient algorithm depending on the specific multi-core system architecture, and the number of cores and message length involved in the collective operation; (5) supporting the automatic performance tuning of the collectives depending on the system and communication parameters; and (6) allowing its integration in any MPJ implementation as it is based on MPJ point-to-point primitives. A performance eval-

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

Published online: 21 August 2010

uation on an InfiniBand and Gigabit Ethernet multi-core cluster has shown that the implemented collectives significantly outperform the original ones, as well as higher speedups when analyzing the impact of their use on collective communications intensive Java HPC applications. Finally, the presented library has been successfully integrated in MPJ Express (http://mpj-express.org), and will be distributed with the next release.

Keywords Message-passing in Java (MPJ) · Multi-core clusters · Scalable collective communication · High performance computing · Performance evaluation

1 Introduction

Java is the leading programming language both in academia and industry environments, and it is an alternative for High Performance Computing (HPC) [1] due to its appealing characteristics: built-in networking and multithreading support, object orientation, automatic memory management, platform independence, portability, security, an extensive API and a wide community of developers, and besides it is the main training language for computer science students. Moreover, performance is no longer an obstacle. Java, in its early days, was severely criticized for its poor computational performance, reported to be within a factor of four of the equivalent Fortran code in [2]. However, currently, thanks to advances in JVMs and Just-In-Time (JIT) compilation, which are able to generate native executable code from the platform independent bytecode, Java performance is around a 30% slower on average than natively compiled languages (e.g., C and Fortran), according to [1] and [3]. This relatively low overhead trades off for the interesting features of Java. However, although this performance gap is relatively small, it can be particularly high for communication-intensive parallel applications when relying on poorly scalable Java communication libraries, which has hindered Java adoption for HPC. Thus, this paper presents a more scalable collectives communication library.

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

The increasing number of cores per system demands efficient and scalable message-passing communication middleware. However, up to now Message-Passing in Java (MPJ) implementations have been focused on providing production-quality implementations of the full MPJ specification, rather than concentrate on developing scalable collective 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. Unfortunately, current MPJ collective libraries show poor scalability [4]. This paper presents a scalable and efficient MPJ collectives library aiming to its integration in MPJ implementations in order to provide higher performance on multi-core clusters.

Although it could seem that the translation of successful research in MPI collectives optimization into the MPJ arena would suffice, the particularities of the Java execution environment pose several additional challenges to the usually complex development of efficient collective operations. Thus, the new collectives library has to cope with the latency *jitter*, the JVM runtime execution behavior, the poor Java high speed network latency, and the restriction to the use of point-to-point operations in the collectives implementation.

MPJ collective primitives show an important variation for the minimum, average and maximum latencies. This variability of their performance results over time is known as *jitter*, or latency variation. One of the most important factors for Java communications *jitter* is the JVM operation, especially the JIT compiler and the JVM runtime execution. The impact of the *jitter* in the overall collectives performance is minimized reducing the synchronization points in their algorithms.

Moreover, the JVM presents several execution modes, such as interpreted bytecode and several levels of native code generation from the Just-In-Time (JIT) compiler. The particular performance of an MPJ collective call significantly depends on the JVM operation, which tends to further optimize commonly used methods with high overheads. Thus, it is quite often to find a collective operation with theoretically higher overhead outperforming a collective call with smaller message payload. This issue can be addressed reusing communication methods, or exploding recursion in order to make them eligible for further optimizations of the JIT compiler. Moreover, continuous changes in the communication protocols and algorithms, quite common in native MPI bindings, are also avoided for this reason.

Furthermore, the lack of efficient high-speed networks support in Java, due to its inability to control the underlying specialized hardware, results in lower performance than MPI, especially for short messages. In fact, MPJ hardly obtains around a 10% of MPI short message performance, whereas it can achieve up to a 90–95% of MPI bandwidth. As a consequence of this, most of the techniques used in MPI to speed up collectives performance, such as message fragmentation and synchronous protocols (rendezvous protocol), are useless. Nevertheless, new optimization techniques arise, such as message aggregation and asynchronous operations (eager protocol).

Finally, MPI implementations, looking for performance, can rely on collective operations implemented natively in the communication hardware or in low-level communication libraries, whereas MPJ libraries are restricted to implement collective algorithms on top of point-to-point Java communication primitives. Although it is possible to rely on native collective methods through JNI, this option presents sev-

eral additional drawbacks associated with the use of native methods (e.g., lack of portability, JVM security and instability issues, and significant JNI copy overhead).

The development of efficient collective operations taking into account these characteristics of the Java execution model constitutes the main research effort carried out in this work.

Furthermore, additional contributions of this paper are focused on: (1) providing multi-core aware collective primitives, through minimizing inter-node communications and favoring multithreading-based solutions; (2) selecting at runtime the most efficient algorithm, based on the number of processes and message size; and (3) supporting an automatic performance tuning process, whose outcome is the generation of an optimal configuration file with the algorithms that maximize the collectives performance in a given system.

The structure of this paper is as follows: Sect. 2 presents MPJ background information. Section 3 introduces the related work. Section 4 describes the design and implementation of the efficient MPJ collectives library, covering in detail the operation of the communication algorithms, their multi-core architecture awareness, and the support for the runtime selection of the collective algorithms that provide the highest performance in a given multi-core system. Section 5 shows the performance results of the implemented library on an InfiniBand and Gigabit Ethernet multi-core cluster. The evaluation consists of a micro-benchmarking of collective primitives, with a special emphasis on analyzing the scalability of the collective operations, as well as a kernel/application benchmarking in order to analyze the impact of the use of the library on their overall performance. Section 6 summarizes our concluding remarks and future work.

2 Message-passing in Java

Message-passing is the most widely used parallel programming paradigm as it is highly portable, scalable and usually provides good performance. It is the preferred choice for parallel programming distributed memory systems such as clusters, which can provide higher computational power than shared memory systems. Regarding the languages compiled to native code (e.g., C and Fortran), MPI is the standard interface for message-passing libraries.

With respect to Java, there have been several implementations of Java message-passing libraries since its introduction [1]. Although initially each project developed their own MPI-like binding for the Java language, current projects generally adhere to one of the two main proposed APIs, the mpiJava 1.2 API [5], which supports an MPI C++-like interface for the MPI 1.1 subset, and the JGF MPJ API [6], which is the proposal of the Java Grande Forum (JGF) [7] to standardize the MPI-like Java API. The collective communication primitives are essential part of the different MPJ APIs, both in terms of number of methods and widespread use.

MPJ libraries can be implemented in two ways: (1) wrapping an underlying native messaging library like MPI through Java Native Interface (JNI); or (2) using a "pure" Java (100% Java) approach, based on RMI or sockets. Each solution fits with specific situations, but presents associated trade-offs. The use of the pure Java approach ensures portability, but it might not be the most efficient solution, especially

in the presence of high-speed communication hardware and when using RMI or Java Message Service (JMS) as these technologies are oriented to distributed computing on loose coupled peers and show high start-up latencies. The use of JNI has portability problems, although usually in exchange for higher performance. With respect to the MPJ collective library implementation, in a wrapper MPJ library it consists of a collection of wrapper classes that rely on an underlying MPI collective library implementation, whereas a pure Java MPJ library requires a whole collectives implementation, usually on top of point-to-point communications. Although the MPJ proposals do not discuss the handling of collective libraries, it is possible to integrate third-party collective libraries in any MPJ project, especially if the collective operations are based on MPJ point-to-point operations.

The mpiJava library [8] is a wrapper implementation which provides efficient communication resorting to an underlying native MPI library, adding a reduced JNI overhead. However, despite its usually high performance, mpiJava currently only supports some native MPI implementations, as wrapping a wide number of functions, especially the collectives, and heterogeneous runtime environments entails an important maintaining effort. Additionally, this implementation presents instability problems, derived from the native code wrapping, and it is not thread-safe, being unable to take advantage of multi-core systems through multithreading.

As a result of these drawbacks, the mpiJava project maintenance has been superseded by the development of MPJ Express [9], a "pure" Java message-passing implementation of the mpiJava 1.2 API specification. MPJ Express is thread-safe and presents a modular design which includes a pluggable architecture of communication devices that allows to combine the portability of the "pure" Java New I/O package (Java NIO) communications (niodev device) with the high performance Myrinet support (through the native Myrinet eXpress—MX—communication library in mxdev device). Furthermore, this project is the most active in terms of adoption by the HPC community, presence on academia and production environments, and available documentation. This project is also stable and publicly available along with their source code at http://mpj-express.org. However, MPJ Express is currently distributed with a poorly scalable collective library that is a limiting factor for its definitive adoption in HPC.

Additional MPJ implementations, such as MPJ/Ibis [10], include their own collective library, although quite often these implementations are incomplete and their primitives implement poorly scalable algorithms, as they usually do not take advantage of non-blocking communications. The aim of this work is to provide MPJ libraries with a portable, efficient and scalable collective library. Moreover, this library has been integrated within MPJ Express, showing significantly higher scalability for MPJ collective communications.

3 Related work

As far as we know, this is the first project devoted exclusively to the optimization of MPJ collective communications, as up to now MPJ library developments have disregarded the development of scalable and efficient MPJ collective primitives. Moreover,

the design and implementation of MPJ collective libraries is usually discussed in the related literature together with their corresponding MPJ projects, as the collectives are essential part of their tightly coupled designs. Therefore, few papers consider MPJ collective communications, although usually without paying enough attention to their significance and impact on Java HPC performance.

Thus, the most relevant related literature in MPJ collective communications comprises the papers that introduce MPJ/Ibis [10], MPJava [11], and Fast MPJ (F-MPJ) [4] projects. With respect to MPJ/Ibis, its collective library only implements one algorithm per primitive. Unfortunately, the selected algorithms are poorly scalable as they are usually based on blocking point-to-point communications (except for Alltoall/Alltoallv). Additionally, MPJava implements a subset of the MPJ collective operations, showing also poor scalability. Their performance results, presented in [11], highlighted for the first time in the MPJ community the importance of choosing the appropriate collective communication algorithm according to the characteristics of the codes being executed and the hardware configuration employed. Finally, our own MPJ implementation F-MPJ [4] includes a scalable collective library implemented on top of point-to-point calls to a low-level communication device. Moreover, F-MPJ collective library implements up to three algorithms per primitive, selected statically (at compile-time).

With respect to the scalability of current MPJ collective libraries, several performance evaluations [4, 9, 12] have pointed out that their results are generally poor due to the use of algorithms that do not take advantage of multi-core architectures and non-blocking communications, in order to exploit data locality and overlap communications, respectively.

The collective library presented in this paper improves current MPJ collective implementations (using different techniques): (1) providing multi-core aware collective primitives; (2) implementing up to six algorithms per collective operation; (3) selecting the most scalable algorithm at runtime depending on the specific multi-core system architecture, and the number of cores and message length involved in the collective operation; (4) supporting, through a portable and transparent mechanism, the automatic performance tuning of the collectives operation depending on the system and communication parameters; and (5) allowing an easy integration with any MPJ implementation as it is based on MPJ point-to-point operations.

Most of the contributions of the implemented library have been motivated by the success of related works in native message-passing libraries, where far more research has been done. Thus, our library has adapted the research in MPI to MPJ, taking into account the specificities of Java: (1) high variations (jitter) in the communication latencies caused by the Java execution environment and not present in MPI. The impact on MPJ can be minimized reducing the synchronization points in the algorithms implemented; (2) different JVM execution modes, such as interpreted bytecode and several levels of native code generation from the JIT compiler. This issue can be addressed reusing communication methods, or exploding recursion, in order to make them eligible for further optimizations of the JIT compiler. Moreover, continuous changes in the communication protocols and algorithms, quite common in native MPI implementations, are also avoided for this reason; (3) high start-up latencies, due to the lack of efficient high-speed networks support. As a consequence of this, most

of the techniques used in MPI to speed up collectives performance, such as message fragmentation and synchronous protocols, are useless. Nevertheless, new optimization techniques arise, such as message aggregation and asynchronous protocols; and (4) the limitation of being based on point-to-point operations, due to the lack of direct support in Java, unlike MPI, of collective operations implemented natively in the communication hardware or in low-level communication libraries.

With respect to the optimization of MPI collective operations, in [13] Chan et al. discuss thoroughly the design and high-performance implementation of collective communications operations on distributed-memory computer architectures. In [14] and [15], two model-based approaches for selecting the communication strategy that better adapts to a particular scenario are presented. In [16] it is suggested the use of a number of different algorithms in order to select the most scalable for a particular message size and number of processes involved in the communication. As it is highly desirable that this selection can be made automatically, the efficiency of this process has been tackled in [17], obtaining less than a 5% performance penalty on average.

Regarding the efficiency of collective communications on multi-core architectures, several research lines have been explored. Thus, the hierarchical approach has provided significant performance increase for the Alltoall operation [18]. An extension of this strategy is a two-level intra-node and inter-node hierarchy [19], but this discrimination of intra-node and inter-node hierarchies scarcely increases performance. In fact, currently most of the overhead is in the inter-node communication, so the optimization of intra-node MPI collective algorithms does not improve much the overall collectives performance. However, the intra/inter-node awareness happens to be the key aspect to be taken into account in collectives performance optimization. Furthermore, the optimization based on hierarchical virtual topologies presented in [20] has achieved significant performance gains owing to the use of cache-aware intra-node communications. Another approach is the maximization of the use of shared memory and the reduction of network communications on shared/distributed memory systems. This strategy usually provides significant performance advantages [21]. Finally, the efficient placement of MPI processes in a multi-core system (runtime process attachment to specific cores) is discussed in [22].

Additional projects involving the optimization of Java collective communications for HPC are CCJ [23] and the Java Adlib collective library [24]. CCJ is an RMI-based Java collective communication library for HPC which implements a simple low-level API. This library is intended to support collective operations in higher-level libraries, such as MPJ. Thus, MPJ/Ibis includes CCJ collective implementations. However, this library is poorly scalable, mainly due to the use of RMI, and lacks popular collective operations such as Alltoall and reduction operations. With respect to Java Adlib collective library, it is a high-level collective communication library primarily focused on HPJava, a data-parallel Java programming environment for HPC, so its applicability to message-passing communications is quite limited.

4 Scalable MPJ collective communications

The main motivation of this work is to implement a scalable and efficient MPJ collective communication library, taking advantage of communications overlapping, multi-

Fig. 1 Broadcast bFT

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

core awareness, and runtime selection of the most appropriate algorithm, which depends on the message size and the number of processes involved in the communication. As MPJ performance heavily depends on the scalability of collective communications, the implemented library is of special interest, especially for its integration in an MPJ implementation whose original collectives library shows poor scalability, such as MPJ Express. Nevertheless, the collectives library presented in this paper can be easily integrated in the remaining MPJ implementations as, aiming at portability, all collective algorithms are implemented using MPJ point-to-point operations (the different APIs show little variation in point-to-point methods).

4.1 Collective communication algorithms

The collective algorithms implemented in our library can be classified in six types, namely Flat Tree (FT) or linear, Four-ary Tree (FaT), Minimum-Spanning Tree (MST), Binomial Tree (BT), Bucket (BKT) or cyclic, and BiDirectional Exchange (BDE) or recursive doubling. These algorithms are thoroughly described in [13].

The simplest algorithm is the FT (Flat Tree), where all communications are performed sequentially. Figures 1 and 2 show 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, respectively. As a general rule, 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, p_i the ith process and `npes` is the number of processes used.

With respect to Gather, the default MPJ Express implementation uses a Flat Tree with non-blocking receptions and blocking sends, whose pseudocode is shown in Fig. 3 (Gather nbFT). Nevertheless, this is an inefficient implementation, as the non-blocking operation cannot overlap any computation/communication. An alternative

Fig. 2 Broadcast nbFT

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

Fig. 3 Gather nbFT

```
if me != root then
    sreq =ISEND (x_me,root);
    WAIT (sreq);
else
    for i=0,...,npes-1 do
        if me != i then
            RECV (x_i,p_i);
```

Fig. 4 Gather nb1FT

```
if me = root then
    for i=0,...,npes-1 do
        if me != i then
            rreq_i =IRECV (x_i,p_i);
    for i=0,...,npes-1 do
        if me != i then
            WAIT (rreq_i);
else
    SEND (x_me,root);
```

Fig. 5 Alltoall nbFT

```
for i=0,...,npes-1 do
    if me != i then sreq_i =ISEND (x_me,i,p_i);
for i=0,...,npes-1 do
    if me != i then RECV (x_i,me,p_i);
for i=0,...,npes-1 do
    if me != i then WAIT (sreq_i);
```

solution, which would provide higher performance, is the call to the non-blocking primitive inside the loop, overlapping communications. Thus, the root process can post all the non-blocking reception calls in advance (see Gather nb1FT pseudocode in Fig. 4). Here the use of non-blocking primitives in the sender side does not increase the communication performance.

There are four variants of the Flat Tree algorithm for the Alltoall primitive, implementing the four available combinations of the use of blocking/non-blocking point-to-point communications for the send and receive operations: using both blocking sends and receives (bFT); using non-blocking sends and blocking receives (nbFT) (Fig. 5); using both non-blocking sends and receives (nb1FT) (Fig. 6); and using blocking sends and non-blocking receives (nb2FT) (Fig. 7).

These four variants allow the selection of the algorithm that maximizes the performance. Although it might seem that the use of both non-blocking sends and receives (nb1FT) is the most scalable solution, this option has associated some drawbacks, such as the contention and congestion that might appear in the underlying communication layer, thus causing serious performance bottlenecks. Therefore, the library implements four variants as this algorithm involves an important number of messages whose performance can vary significantly among different communication systems. Thus, each of the four alternative implementations is expected to maximize the performance on a particular range of scenarios.

Four-ary Tree (FaT) algorithms configure a tree in which each node has four successors at the most. Thus, the communication operation consists of traversing this "four-ary" tree. Figure 8 presents an example for the Broadcast, where the boxes rep-

Fig. 6 Alltoall nb1FT

```
for i=0,...,npes-1 do
    if me != i then
        sreq_i = ISEND (x_{me,i}, p_i);
    if me != i then
        rreq_i = IRECV (x_{i,me}, p_i);
for i=0,...,npes-1 do
    if me != i then WAIT (sreq_i);
for i=0,...,npes-1 do
    if me != i then WAIT (rreq_i);
```

Fig. 7 Alltoall nb2FT

```
for i=0,...,npes-1 do
    if me != i then
        SEND (x_{me,i}, p_i);
    if me != i then
        rreq_i = IRECV (x_{i,me}, p_i);
for i=0,...,npes-1 do
    if me != i then
        WAIT (rreq_i);
```

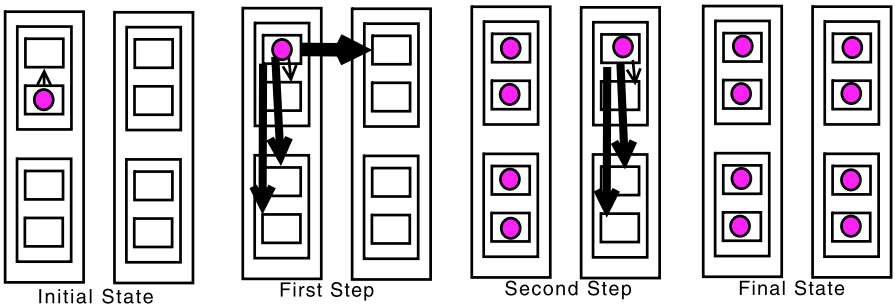

Fig. 8 Broadcast Four-ary Tree (FaT)

resent, for each state, two nodes, with two associated processors per node, and two cores per node. Additionally, the thickness of the arrow symbolizes the communication cost of a point-to-point communication, i.e., the thicker the arrow the more communication cost. From now on this representation would be used for the description of the communication operation of the following algorithms.

MST algorithms present a more complex operation. Thus, the total number of processes involved in the operation is recursively divided into two subsets. After each division the root process sends its message to a process of the other subset, which will become the root for its subset. This task is recursively repeated until all processes have performed their expected operations. As MST algorithms minimize the communications between distant processes, they significantly improve the collectives performance on multi-core clusters. Figures 9 and 10 show MST operations for a Broadcast and Gather communication patterns, respectively.

With respect to the BiDirectional Exchange (BDE) or recursive doubling algorithm, it subdivides the process set just like MST, but the new subsets have to be of

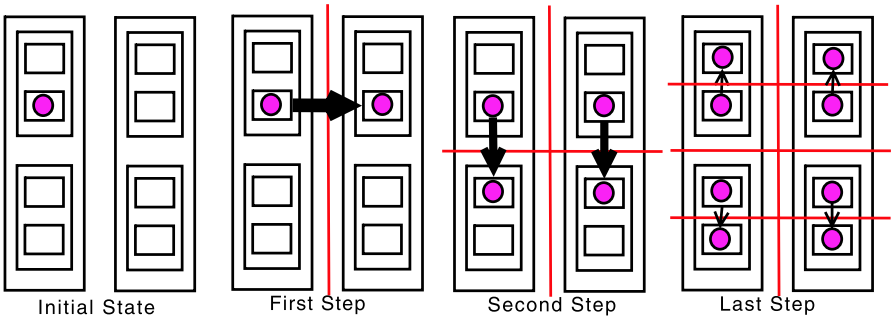

Fig. 9 Broadcast MST

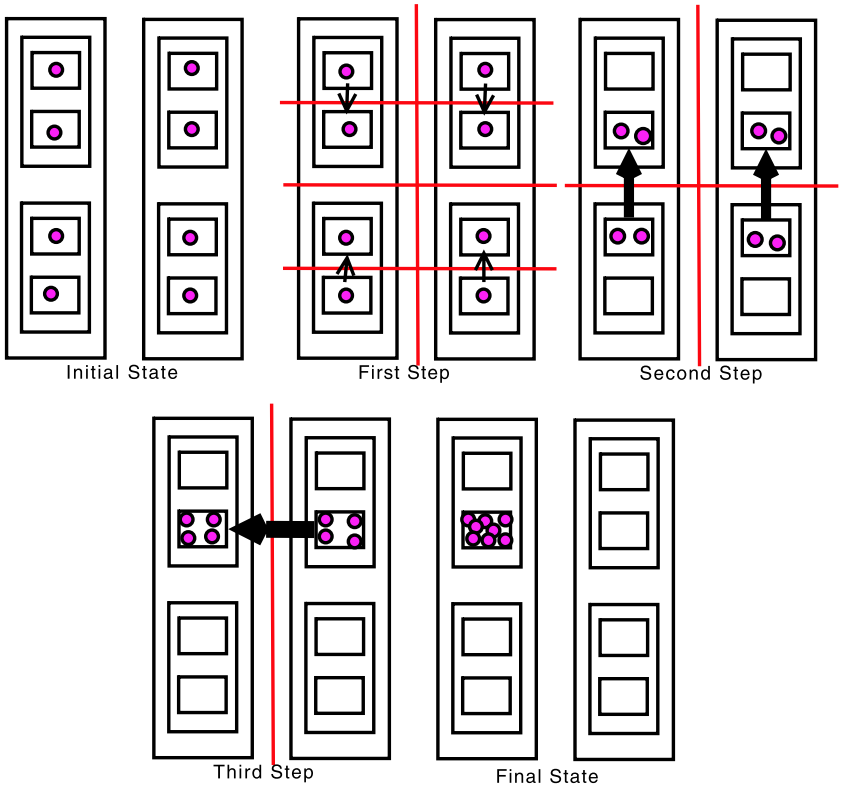

Fig. 10 Gather MST

the same size. Therefore, BDE requires that the number of processes be a power of two for its correct work. Regarding the communications between two subsets, each process selects a counterpart process from the other subset in order to perform the communication required by the algorithm, as shown in Fig. 11, for an Allgather communication pattern.

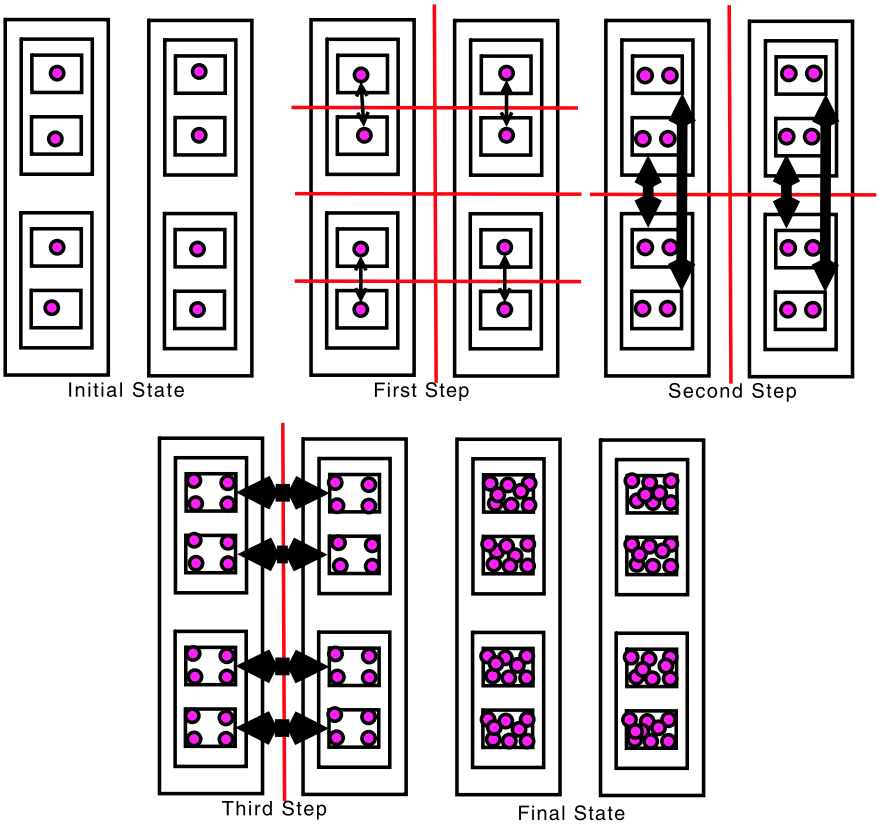

Fig. 11 Allgather BDE

The BDE algorithm can also be seen as a particular case of BKT or Bucket algorithm. In BKT all processes are organized like a ring, and they send at each step data to the process at their right. Thus, data eventually arrives to all nodes. BKT does not pose restrictions on the number of processes communicating. Figure 12 shows an example of its operation for the Allgather primitive. Looking for potential optimizations, our library implements a variant (nbBKT) of this algorithm, posting all receive requests at BKT start-up. Thus, it is avoided the buffering overhead caused by the arrival of a message whose reception request has not already been posted. Then, the communication overlapping is achieved through non-blocking send calls.

The main difference between BDE and BKT for a given scenario consists in the number of steps involved in the communication, showing linear and logarithmic complexities for BKT and BDE, respectively. Moreover, the message recursively increases for BDE, while it remains constant for BKT. Regarding the suitability of these algorithms, a shared memory environment would benefit from the parallelism in BKT communications, whereas the aggregation of small messages into a larger one in BDE significantly improves performance on high latency networks. Finally, these algorithms are much more scalable than the Flat Tree ones.

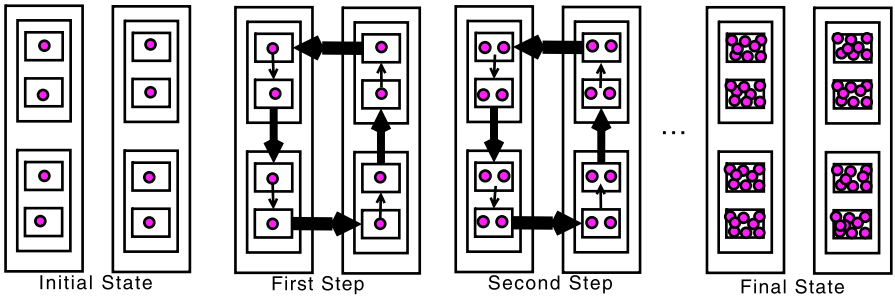

Fig. 12 Allgather BKT

Table 1 Algorithms implemented in the MPJ collective library

Primitive	MPJ Express	New MPJ Collectives Library
Barrier	Gather + Bcast	nbFTGather + bFaTBcast, Gather + Bcast, BT
Bcast	bFaTBcast	bFT, nbFT, bFaTBcast, MST
Scatter	nbFT	nbFT, MST
Scatterv	nbFT	nbFT, MST
Gather	nbFT	bFT, nbFT, nb1FT, MST
Gatherv	nbFT	bFT, nbFT, nb1FT, MST
Allgather	nbFT, BT	nbFT, BT, nbBDE, bBKT, nbBKT, Gather + Bcast
Allgatherv	nbFT, BT	nbFT, BT, nbBDE, bBKT, nbBKT, Gather + Bcast
Alltoall	nbFT	bFT, nbFT, nb1FT, nb2FT
Alltoallv	nbFT	bFT, nbFT, nb1FT, nb2FT
Reduce	bFT	bFT, nbFT, MST
Allreduce	nbFT, BT	nbFT, BT, bBDE, nbBDE, Reduce + Bcast
Reduce-Scatter	Reduce + Scatterv	bBDE, nbBDE, bBKT, nbBKT, Reduce + Scatter
Scan	nbFT	nbFT, lineal

Table 1 presents a list of the collective algorithms implemented for MPJ Express and the new collectives library. The implementation variants are correlatively numbered (e.g., nb1FT and nb2FT are variants of nbFT, as it was previously explained). As can be seen, the higher number of algorithms implemented (55 vs. 17) allows a wider selection, being feasible to take more advantage of the underlying hardware.

4.2 Thread-based collective primitives

Current MPJ libraries usually implement intra-node communications through inter-process mechanisms such as sockets and System V IPC shared memory. In fact, only MPJ Express includes a thread-based shared memory communication device, namely `smpdev` [25], which takes advantage of Java multithreading, although its scalability is limited due to costly synchronization overheads. Moreover, the combination of two or more communication devices (e.g., `smpdev` and `niodev`) is not yet supported in MPJ Express, so `smpdev` cannot be used in a hybrid shared/distributed memory

architecture, such as a cluster of multi-core nodes. In this kind of architectures the approach that can take full advantage of the underlying hardware is the simultaneous use of the message-passing paradigm together with thread-based solutions. This hybrid message-passing/multithreading approach requires a thread-safe MPJ implementation, and MPJ Express is claimed to support the highest level of thread safety, `MPI_THREAD_MULTIPLE`. Thus, MPJ Express is usually the preferred choice for hybrid message-passing and multithreading programming such as threads or Java OpenMP implementations (e.g., JOMP) [9].

However, this hybrid programming approach, although it can provide good performance, has as main drawback the fact that it requires the use of two programming paradigms, increasing the complexity of the parallel programming. We believe that only one programming model should be used, and currently message-passing can scale further than the shared memory programming model. Thus, message-passing libraries should support intra-node communications as shared memory transfers, a feature currently not implemented by any MPJ library. Nevertheless, in order to explore this possibility, the new collectives library presented in this paper has been extended in order to support this approach. This thread-based message-passing collectives library will serve as a proof of concept of the performance benefits that can be obtained following the aforementioned approach.

This library extension has been implemented selecting one thread per node (`rootThread`) in order to be in charge of the inter-node message-passing communications. Additionally, this thread will also serve as root thread for the intra-node execution of the collective operation. Thus, the thread-safety requirement of the MPJ implementation for our prototype library is limited to the support of the `MPI_THREAD_FUNNELED` level, which means that a process may be multi-threaded but only the thread that initialized MPJ makes MPJ calls. Finally, in order to exploit the data locality and affinity, each thread defines its storage space in its TLA (Thread Local Area).

Figure 13 presents the algorithm of the thread-based Broadcast, where `x` is the message, `rootProcess` is the root process, `rootThread` is the root thread within each process, `myThreadRank` is the rank of each thread within each process, and `NUM_THREADS` is the number of threads within one process.

The data access synchronization is controlled by `ready` and `done`, two `AtomicInteger` objects from the concurrency framework. An `AtomicInteger` is an int value (counter) that is updated atomically through several methods, three of them used in our library: `compareAndSet(int expect, int update)`, which atomically sets the counter to the given updated value if the current value equals the expected one; `incrementAndGet()`, which atomically increments by one the counter; and `get()`, which gets the current counter value.

This thread-based algorithm first broadcasts the data among the processes, through the `rootThread` invocation in all threads. After this process-level operation, a thread-level intra-node broadcast is performed. This operation consists in that all the threads but the `rootThread` copy in parallel the broadcast message into thread local variables. This local copy of the data is needed in order to exploit data locality, avoiding bottlenecks in shared memory accesses, as well as cache invalidation, quite common performance penalties in multi-core systems.

```
AtomicInteger ready = new AtomicInteger(0);
AtomicInteger done = new AtomicInteger(0);
if myThreadRank = rootThread then
    BCAST (x,rootProcess);
    while not ready.compareAndSet(NUM_THREADS-1,0) do WAIT ();
    while not done.compareAndSet(NUM_THREADS-1,0) do WAIT ();
else
    ready.incrementAndGet();
    while ready.get()>0) do WAIT ();
    System.arraycopy(x,0,TLA[myThreadRank].x,0,x.length);
    done.incrementAndGet();
    while done.get()>0) do WAIT ();
```

Fig. 13 Threaded Broadcast of **x** with *NUM_THREADS* threads per process

4.3 Portability and automatic selection of algorithms

One of the aims of this work is to provide a portable library that can be easily integrated in any MPJ implementation. In order to achieve this goal, the communication algorithms have been implemented in the Intracomm class, which contains the collectives implementation according to the main MPJ proposals. In order to avoid dependencies that would break the portability of the developed library, all collective algorithms are implemented using MPJ point-to-point operations, which show almost no variation among the different APIs. In fact, initially the library was implemented in MPJ Express, with mpiJava 1.2 API, and in order to support the JGF MPJ API, implemented by MPJ/Ibis, the only changes needed were renaming the MPJ package (e.g., in data types MPI.BYTE → MPJ.BYTE), as well as some methods (e.g., Send → send and Recv → recv). Moreover, no assumption has been made about the concrete operation of the data transfers, which are MPJ implementation dependent (e.g., they can be synchronous or asynchronous).

Additionally, the objective of the easy integration has been fulfilled, as the developed library has been successfully incorporated into MPJ Express, which has been selected for the integration due to its popularity, active development, and, especially, for being distributed with a poorly scalable collective communications library. Moreover, the new collectives library is fully transparent to the user. Finally, the portability of the collectives implementation is assessed through its support by F-MPJ and MPJ/Ibis.

The runtime selection of the collective algorithm that provides the highest performance in a given multi-core system, among the several algorithms available, is based on the message size and the number of processes. The definition of a threshold for each of these two parameters allows the selection of up to four algorithms per collective primitive. Moreover, these thresholds can be configured for a particular system by means of an autotuning process, which obtains an optimal selection of algorithms, based on the particular performance results on a specific system and taking into account the particularities of the Java execution model.

This selection is stored in a configuration file (`collectives.properties`) that is loaded by a static initializer at MPJ initialization. This configuration file contains information about which algorithm has to be selected depending on the message length and the number of processes involved in the communication. Finally, the use

of the selected algorithms is fully transparent to the user. Thus, if the `collectives.properties` file exists, the MPJ implementation will select the appropriate collective algorithm, otherwise the original one will be used.

The autotuning process consists of the execution of our own MPJ collectives micro-benchmark suite [26], the gathering of their experimental results, and finally the generation of the configuration file that contains the algorithms that maximize performance. The performance results have been obtained on a number of processes power of two, up to the total number of cores of the system, and for message sizes power of two. The parameter thresholds, which are independently configured for each collective, are those that maximize, in relative terms, the performance measured by the micro-benchmark suite. Moreover, this autotuning process is done once per system configuration, previous to the run of the applications. A dynamic runtime system approach would present initially several drawbacks, such as a higher overhead of the algorithm selection process, and probably the use of a wider range of algorithms which is something penalized by the JVM JIT compiler, which in turn benefits commonly reused methods.

Figure 14 depicts a representative example of the performance results gathered, and the selected algorithms (labeled *Autotuning selection*). The autotuning process for the Allreduce primitive involves the evaluation of all the Allreduce algorithms, as well as all the combinations of the Broadcast and Reduce algorithms (an Allreduce algorithm consists of the combination of a Broadcast followed by a Reduce). However, for clarity purposes, only the most representative algorithms (and their combinations) are shown in the graphs. As can be seen in the graphs, the selected message size threshold is 32 KB. At this point the select algorithm changes, although sometimes there is no variation in the selected algorithm that depends on the message size. The combination of MST algorithms have been selected for messages larger than 32 KB, the combination of bFTReduce and nbFTBroadcast maximizes the short message performance on 4 processes, whereas the combination of bFTReduce and MSTBroadcast maximizes the short message performance on 32 processes.

Additionally, based on the performance results, it has been determined the number of processes threshold, which has been set to 8, the number of nodes of the testbed. Thus, algorithms that exploit the use of more than one process per node are used from 8 processes on, whereas algorithms that maximize performance in a scenario of dedicated network access are used for up to 8 processes.

Table 2 presents the information contained in the optimum configuration file for the multi-core cluster used in the experimental evaluation presented in this paper (Sect. 5).

Finally, the overhead of the autotuning process and the communication algorithm selection is reduced, as the autotuning process does not interfere with the execution of MPJ applications and the selection of algorithms is done through a light mapping method based on the number of processes and message size of a particular data transfer. This mapping is backed by a Java HashMap with approximately 60 keys and whose access overhead is around 10 microseconds as it handles the different algorithms and scenarios represented by integer codes. Moreover, the access to the HashMap is not synchronized, as the algorithms do not change dynamically.

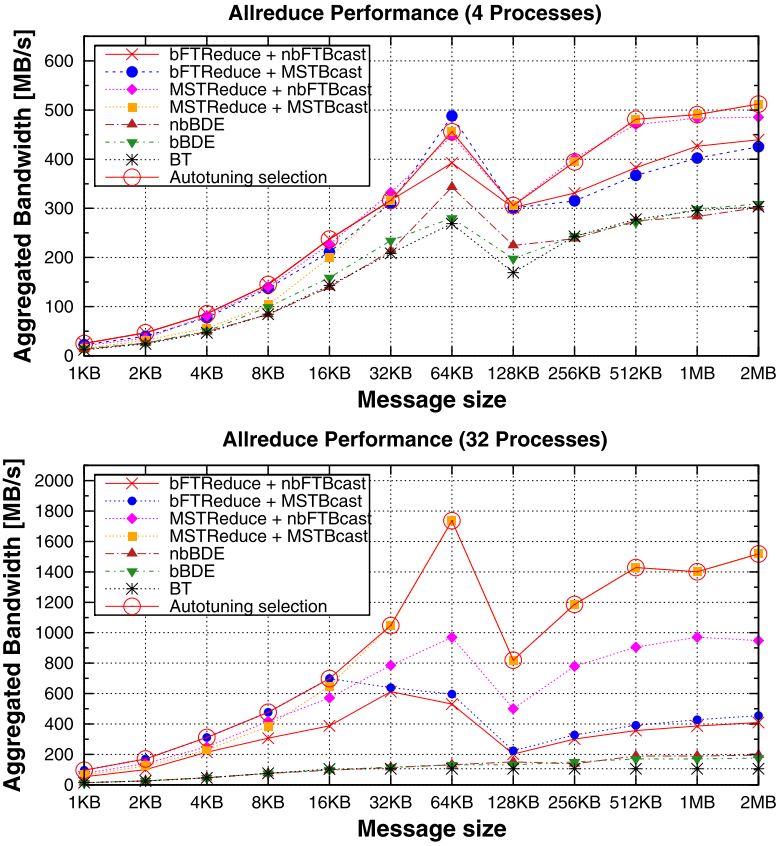

Fig. 14 Performance-based selection of Allreduce algorithms in the autotuning process

5 Performance evaluation

The evaluation presented in this section consists of a micro-benchmarking of collective primitives (Sect. 5.2), as well as an analysis of the impact on the overall performance of the use of the developed collectives library on three representative MPJ codes (Sect. 5.3).

5.1 Experimental configuration

The evaluation of the developed library has been carried out in a cluster which consists of 8 nodes, each of them with 8 GB of RAM and 2 Intel Xeon E5520 quad-core Nehalem processors, that owing to the HyperThreading (HT), when enabled, can run 16 processes per node concurrently. The interconnection networks are InfiniBand (16 Gbps of maximum theoretical bandwidth), with OFED driver 1.4.2, and Gigabit Ethernet (1 Gbps). The OS is CentOS 5.2, the C compiler Intel icc 11.1, the JVM Sun 1.6.0_13, and the message-passing libraries are MPJ Express 0.27 and Open-MPI 1.3.3. The performance results on this system have been obtained running up

Table 2 Example of configuration file for selection of collective algorithms

Primitive	Short message/ small number of processes	Short message/ large number of processes	Long message/ small number of processes	Long message/ large number of processes
Barrier	nbFTGather + bFatBcast	nbFTGather + bFatBcast	Gather + Bcast	Gather + Bcast
Bcast	nbFT	MST	MST	MST
Scatter	nbFT	nbFT	nbFT	nbFT
Gather	nbFT	nbFT	MST	MST
Allgather	Gather + Bcast	Gather + Bcast	Gather + Bcast	Gather + Bcast
Alltoall	nb2FT	nb2FT	nb2FT	nb2FT
Reduce	nbFT	nbFT	MST	MST
Allreduce	Reduce + Bcast	Reduce + Bcast	Reduce + Bcast	Reduce + Bcast
Reduce-Scatter	bFTReduce + nbFTScatterv	bFTReduce + nbFTScatterv	BDE	BDE
Scan	lineal	lineal	lineal	lineal

to 128 processes, distributing them evenly among the different nodes (e.g., for 128 processes, 16 processes are run per node), except for the thread-based collectives, which use one process per node, and 16 threads per process (hence 128 threads).

5.2 Micro-benchmarking of MPJ collective primitives

Figure 15 presents the aggregated bandwidth results obtained by four representative MPJ collective operations, Broadcast, Allgather, Reduce (sum) and Allreduce (sum), with 128 processes using the thread-based proposed collectives (labeled "New Threaded"), the proposed collectives ("New Colls"), and the original MPJ Express collectives ("Original Colls"). Furthermore, the performance results have been measured using both InfiniBand (with IPoIB, which emulates TCP/IP in order to support Java over InfiniBand) and Gigabit Ethernet as interconnection networks. In order to micro-benchmark collective operations, our own MPJ micro-benchmark suite [26], similarly to Intel MPI Benchmarks, has been used due to the lack of suitable micro-benchmarks for MPJ evaluation. The aggregated bandwidth metric has been selected as it takes into account the global amount of data transferred, generally *Message size* $*$ *Number of processes*. The transferred data are byte arrays, avoiding serialization (the process of transforming an object into a byte array for communication) in order to present the collectives performance without incurring in additional overheads that would distort the analysis of the results. The original MPJ Express collective primitives use the algorithms listed in Table 1 (column MPJ Express), whereas the new collective library uses the algorithms that maximize the performance on this multi-core cluster. Table 3 presents a relevant summary of the configuration file for clarity purposes.

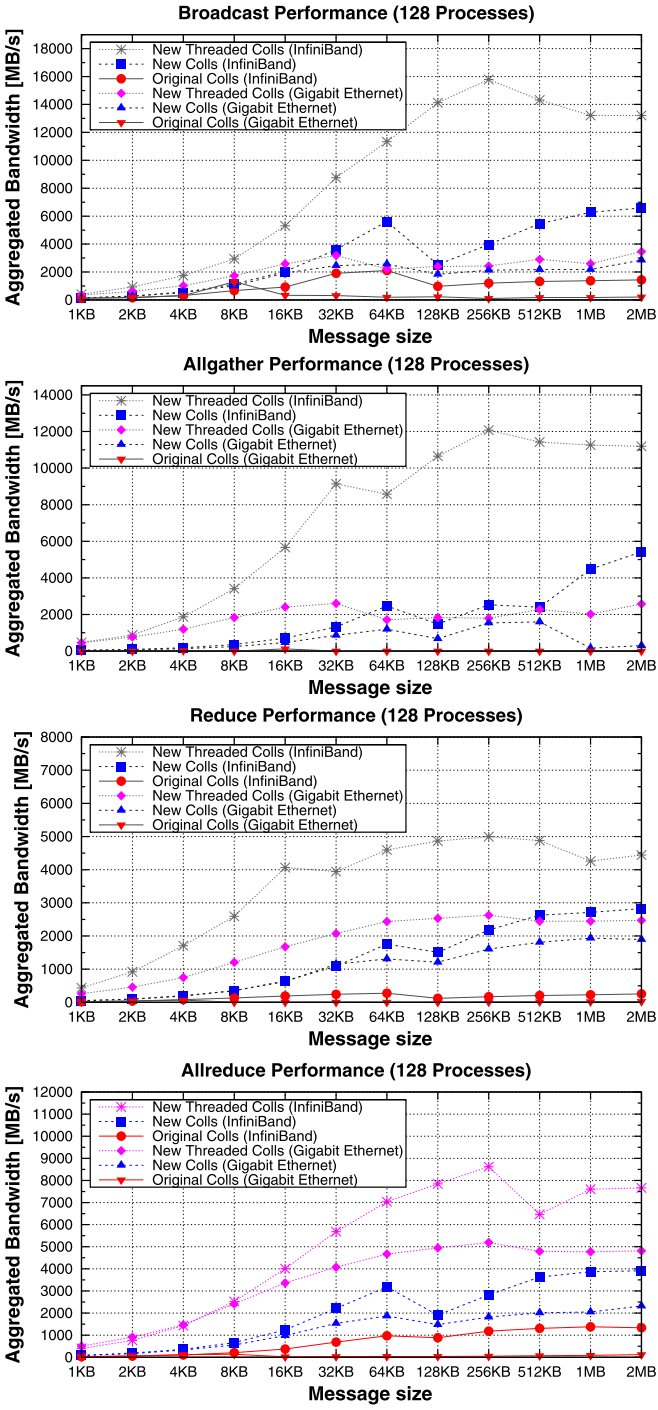

Fig. 15 MPJ collective primitives performance on the Nehalem multi-core cluster

Table 3 Algorithms that maximize performance on the Nehalem multi-core cluster

Primitive	Short message/ small number of processes	Short message/ large number of processes	Long message/ small number of processes	Long message/ large number of processes
Bcast	nbFT	MST	MST	MST
Allgather	nbFTGather + nbFTBcast	nbFTGather + MSTBcast	MSTGather + MSTBcast	MSTGather + MSTBcast
Reduce	bFT	bFT	MST	MST
Allreduce	bFTReduce + nbFTBcast	bFTReduce + MSTBcast	MSTReduce + MSTBcast	MSTReduce + MSTBcast

The presented results (Fig. 15) show that the new library significantly outperforms the original MPJ Express collective library, especially when using the new thread-based implementations and for long messages and Gigabit Ethernet. The main reason of this performance improvement is the efficient exploitation of thread-based shared memory transfers, the use of multi-core aware algorithms, such as MST, which implement communication patterns that reduce the number of costly network communications while taking advantage of the high bandwidth of intra-node transfers. Thus, whereas the Four-ary Tree used in MPJ Express Broadcast needs 112 inter-node (network) transfers for a 128 process operation, the MST Broadcast only requires 7 communications of this type. These numbers of inter-node transfers have been determined with the default mapping provided by MPJ Express runtime, which consists of an even distribution of the processes among the cluster nodes, trying to maintain the adjacency of the process ranks. Although it is possible to define alternative mappings that would increase Four-ary Tree broadcast performance, it is quite likely that they would impact negatively the overall applications and communication algorithms performance, which usually exploit the default mapping of MPJ Express.

Additionally, the Flat Tree-based algorithms implemented in MPJ Express for Allgather, Reduce and Allreduce (see Table 1) are an important performance bottleneck when using a blocking version and an important number of processes (e.g., 128 process Reduce results, both for InfiniBand and Gigabit Ethernet). Moreover, these micro-benchmarks can exhaust the available memory, swapping memory in some cases, and therefore showing poor performance (e.g., original Allgather on Gigabit Ethernet with 128 processes, see Fig. 15), and even hanging the JVMs (e.g., original Allgather on InfiniBand with 128 processes, so its results cannot be taken). The reason is that each node, with 8 GB RAM available, is running up to 16 JVMs and the faster the communications, the higher the memory being consumed. The new collectives library does not show these issues in our testbed.

With respect to the impact of the interconnection network on the overall results, there are significant long message performance differences between InfiniBand and Gigabit Ethernet results, based on the performance achieved by a representative 1 MB point-to-point transfer on this cluster, 115 MB/s over Gigabit Ethernet and 448 MB/s over InfiniBand. However, short message performance is quite similar as the lack of direct InfiniBand support in Java requires the use of the IPoIB TCP emulation,

which shows short message performance similar to TCP/IP over Gigabit Ethernet. Alternatives that would improve MPJ performance on InfiniBand are the support in MPJ Express of SDP (Sockets Direct Protocol), a high performance sockets implementation on InfiniBand, or the development of a new communication device for MPJ Express based on IBV (InfiniBand Verbs), the open-source low-level InfiniBand communication driver, which would significantly outperform IPoIB and SDP MPJ Express support.

With respect to the threaded collectives, it is noticeable that they achieve higher performance gains for data movement primitives, Broadcast and Allgather, than for reduction operations, as the latter ones involve computation which does not take special advantage from a multithreading implementation. Moreover, the benefits of this implementation are usually higher on InfiniBand than on Gigabit Ethernet, as the lower the inter-node overhead, the higher the impact of intra-node optimizations.

Moreover, it has been evaluated the performance scalability of the previously selected representative collective primitives. Figures 16 and 17 present the aggregated bandwidth for Broadcast, Allgather, Reduce (sum) and Allreduce (sum) for representative medium (32 KB) and long (1 MB) message sizes, respectively. These results confirm that the developed library presents significantly better scalability than the MPJ Express original collectives library, especially for the threaded implementation, for long messages and for the Allgather.

Finally, it is noticeable the significant performance increase for the new library when using 128 processes (HT enabled) compared to the use of 64 processes (HT disabled), especially the threaded collectives on InfiniBand, which means that this library implements scalable algorithms and also is taking advantage of the use of HT through communications overlapping on this scenario.

As a global conclusion of the micro-benchmarking analysis, it can be said that the developed library significantly improves MPJ collectives performance due to the implementation of more efficient and scalable algorithms for multi-core architectures. These algorithms take advantage of the communications hardware and shared memory transfers (the threaded version). In fact, the new library can achieve significantly higher performance on a Gigabit Ethernet cluster than the original MPJ Express one on an InfiniBand cluster. However, most of the benefits of the implemented library come also from the automatic selection of the algorithm that maximizes the performance through the best adaptation of the communication pattern to the underlying architecture. As the library includes up to six algorithms per collective primitive, this feature is of special interest in a scenario of increasing complexity in the architecture of current systems.

5.3 MPJ kernel/application performance analysis

The impact of the use of the developed library on representative MPJ kernels and applications is analyzed in this subsection. Two codes from the NAS Parallel Benchmark Suite (NPB-MPJ) [27] have been evaluated: IS, an Integer Sort kernel, and FT, an FFT kernel implementation. Moreover, jGadget [28], the MPJ implementation of the Gadget cosmology simulation application, has also been analyzed. These MPJ codes have been selected as previous analyses of their performance showed

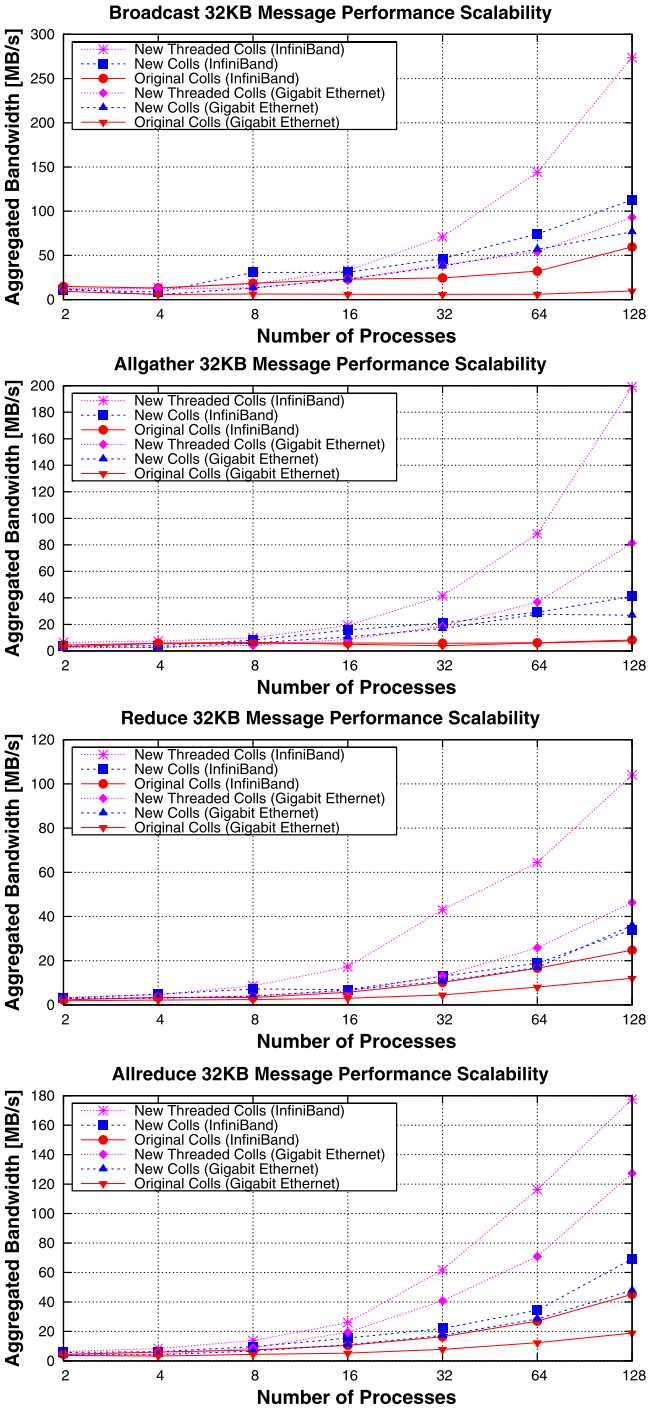

Fig. 16 MPJ collective primitives 32 KB message scalability (Xeon 5520 Nehalem cluster)

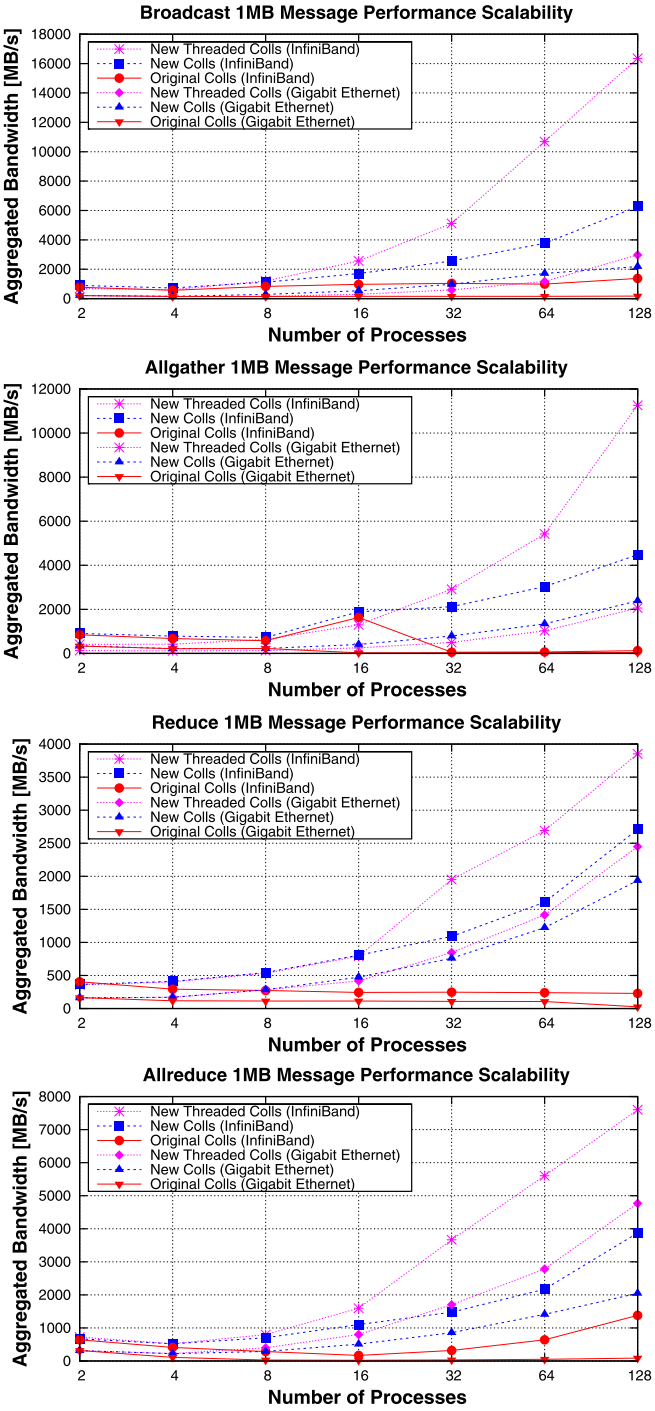

Fig. 17 MPJ collective primitives 1 MB message scalability (Xeon 5520 Nehalem cluster)

poor scalability with MPJ Express due to the collective primitives overhead (Alltoall operations for NPB-MPJ FT, and Allreduce for NPB-MPJ IS and jGadget) [3, 27]. In fact, these are communication-intensive codes, where additional factors, other than communications performance, are usually negligible (e.g., data locality exploitation), except the impact of HT on performance.

Furthermore, in order to ease the analysis of the results, the speedups of the MPI implementations of NPB IS, NPB FT and Gadget are also presented. As MPI collectives are highly optimized, the speedups obtained with them can serve as reference points for assessing the quality of the new collective library. Thus, the impact on performance of the new collectives can be compared in terms of a highly optimized counterpart implementation.

Figure 18 presents the performance of the selected codes in terms of speedup in order to allow a more straightforward analysis of the results as the sequential performance of Java and C/Fortran (C for IS and Gadget, and Fortran for FT) on the testbed differ (Java is around a 30% slower). Thus, it is possible to focus the analysis on the performance scalability, derived mainly from the collectives scalability, thus discarding the differences in sequential performance which can be due to the maturity of the code and the compilers/runtime, as well as the use of specialized numerical libraries (e.g., an FFT parallel implementation) looking for performance.

Although the Nehalem cluster has 64 cores, Hyper-Threading (HT) allows the simultaneous running of 128 processes. The use of HT improves collectives primitives performance, as already shown in the micro-benchmarking results, since they benefit from a certain degree of communications overlapping. Nevertheless, only CPU-bound Java parallel applications with little communications can take advantage of HT. The reason is that, according to our own performance evaluation on our testbed, HT increases Java performance around a 7% on average, obtaining the best results with CPU-bound codes. However, the scalability of parallel applications greatly depends on the communications overhead, so only CPU-bound Java parallel applications with little communications are likely to benefit from HT. Therefore, the communication-intensive message-passing codes evaluated in this subsection are not likely to take advantage of 128 processes, but these results are useful in order to analyze the performance degradation shown, which turned out to be mitigated by the use of the new collectives library proposed in this paper.

Figure 18 (first two graphs) shows the NPB IS and FT results for the Class C workload. Unfortunately, currently MPJ Express does not support the use of the threaded new collectives without re-implementing the user codes. However, this support of threaded intra-node transfers should be transparent to the user, which means that the MPJ library should support this feature. Thus, the following figures do not present results obtained with the threaded collectives.

With respect to the measured performance, as IS is a quite communication-intensive code, with continuous Allreduce and point-to-point communication operations, its scalability is extremely low. In fact, the speedups are below 10 on InfiniBand, taking advantage only of the use of up to 32 processes, even MPI. Here the optimization of the new collective library is limited to the Allreduce operation, showing slightly better performance than the original library on InfiniBand.

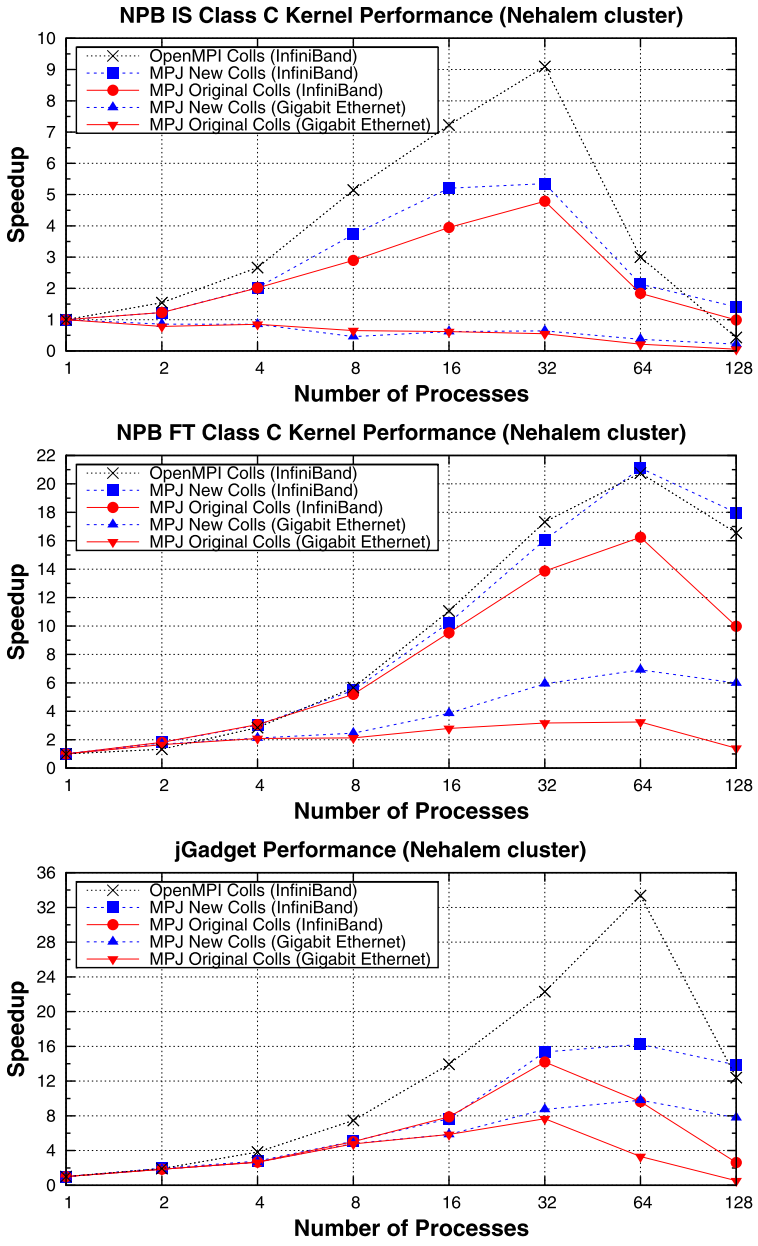

Fig. 18 MPJ kernel/application performance on the Xeon 5520 Nehalem cluster

With respect to FT, this kernel presents better scalability, with speedups of up to 21 on InfiniBand. Here the new library significantly improves the MPJ Express original one, especially on Gigabit Ethernet, and achieves similar performance as MPI. The communication overhead of this kernel is heavily based on the Alltoall primitive,

whose algorithm is a non-blocking Flat Tree both for the original MPJ Express library and the new one. Nevertheless, the new library includes alternative implementations of this algorithm that allow the reported performance improvement.

Last graph in Fig. 18 presents jGadget results from running a two million particles cluster formation simulation. Here the developed collectives library outperforms the original one on 64 and 128 processes, whereas MPI Gadget shows good scalability up to 64 processes (it seems that HT penalizes this implementation).

Additionally, the implemented collective library has been evaluated on a scenario, a supercomputer, which shows significantly higher intra-node and InfiniBand communication bandwidths, in order to assess if the scalability of the codes under evaluation is bounded by the communication collective library or by the underlying communication hardware.

Thus, NPB FT, NPB IS and jGadget have been evaluated on the Finis Terrae supercomputer [29], ranked #427 in November 2008 TOP500 list [30] (14 TFlops), an Itanium2 (IA64) Linux multi-core cluster which consists of 142 HP Integrity rx7640 nodes, each of them with 16 Montvale Itanium2 (IA64) cores at 1.6 GHz and 128 GB of memory, interconnected by InfiniBand (16 Gbps). The OS is SUSE Linux Enterprise Server 10, the JVM is BEA JRockit 5.0 (R27.5), the C compiler is Intel icc 10.1.012, and the MPI is HP MPI 2.2.5.1, with InfiniBand and shared memory communications support. The performance results on this system have been obtained running up to 256 processes on 64 nodes, following an even distribution of the workload (each node runs up to 4 processes).

Figure 19 shows the counterpart performance results of the previously evaluated codes, using the same workloads. The use of this system allows these kernels/application to achieve higher performance, approximately doubling the speedups obtained on the Nehalem cluster. Here the performance results obtained by the new collectives library are relatively close to those of MPI, helping to bridge the performance gap between Java and native languages. The reason for this behavior is that the Finis Terrae achieves especially high intra-node and inter-node communication bandwidths. Additionally, the benefits of using the new collective library are lower in the Finis Terrae than on the Nehalem cluster. However, this confirms what it has been already observed for the new collective library, the lower the scalability and performance provided by the communication hardware, the higher the relative performance benefits achieved.

6 Conclusions

This paper has presented a scalable and efficient MPJ collective communication library for parallel computing on multi-core architectures. The increase in the number of cores per system demands languages with built-in multithreading and networking support, such as Java, as well as scalable and efficient communication middleware that can take advantage of multi-core systems. The new collectives library efficiently exploits multi-core clusters through intra/inter-node awareness and providing a thread-based MPJ collectives implementation. Additionally, the developed library transparently provides Java message-passing applications with several multi-core aware algorithms per collective primitive that can be selected automatically at

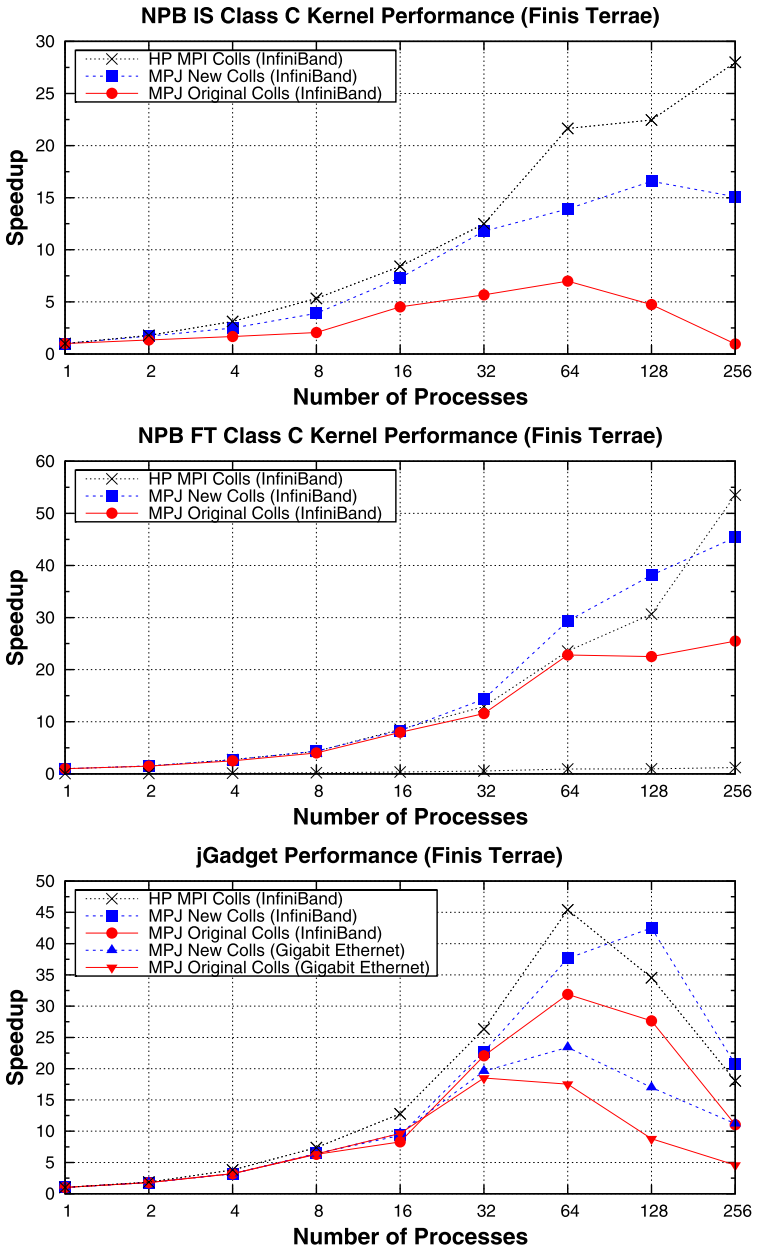

Fig. 19 MPJ kernel/application performance on the Finis Terrae

runtime, depending on relevant parameters, in order to increase communications performance.

The performance evaluation of the collective library on an InfiniBand and Gigabit Ethernet Nehalem multi-core cluster has shown that the implemented collectives

present significantly higher performance than the original ones, especially the multithreaded MPJ approach, as well as higher speedups when analyzing the impact of their use on collective communication intensive Java HPC applications. Additionally, the scalability of the new collectives library has been evaluated on the Finis Terrae supercomputer, showing speedups close to those of MPI using up to 256 cores. Moreover, it has been experimentally assessed that the lower the scalability and performance provided by the communication hardware, the higher the relative performance benefits achieved by the new collective library. Thus, our collectives library can contribute significantly to bridge the performance gap between Java and native languages in HPC.

Acknowledgements This work was funded by the Ministry of Science and Innovation of Spain under Project TIN2010-16735 and an FPU grant AP2009-2112, and by the Xunta de Galicia under Project PGIDIT06PXIB105228PR, the Consolidation Program of Competitive Research Groups and Galician Network of High Performance Computing. We gratefully thank CESGA (Galicia Supercomputing Center, Santiago de Compostela, Spain) for providing access to the Finis Terrae supercomputer.

References

1. Taboada GL, Touriño J, Doallo R (2009) Java for high performance computing: assessment of current research and practice. In: Proc 7th int conf on principles and practice of programming in Java (PPPJ'09), Calgary, Canada, pp 30–39
2. Blount B, Chatterjee S (1999) An evaluation of Java for numerical computing. Sci Program 7(2):97–110
3. Shafi A, Carpenter B, Baker M, Hussain A (2010) A comparative study of Java and C performance in two large-scale parallel applications. Concurr Comput, Pract Exp 15(21):1882–1906
4. Taboada GL, Touriño J, Doallo R (2010) F-MPJ: scalable Java message-passing communications on parallel systems. J Supercomput (in press)
5. Carpenter B, Fox G, Ko S-H, Lim S, mpiJava 1.2: API specification. http://www.hpjava.org/reports/mpiJava-spec/mpiJava-spec/mpiJava-spec.html [Last visited: March 2010]
6. Carpenter B, Getov V, Judd G, Skjellum A, Fox G (2000) MPJ: MPI-like message-passing for Java. Concurr Comput Pract Exp 12(11):1019–1038
7. Java Grande Forum. http://www.javagrande.org [Last visited: March 2010]
8. Baker M, Carpenter B, Fox G, Ko S, Lim S (1999) mpiJava: an object-oriented Java interface to MPI. In: Proc 1st int workshop on Java for parallel and distributed computing (IWJPDC'99), LNCS, vol 1586, San Juan, Puerto Rico, pp 748–762
9. Shafi A, Carpenter B, Baker M (2009) Nested parallelism for multi-core HPC systems using Java. J Parallel Distrib Comput 69(6):532–545
10. Bornemann M, v. Nieuwpoort RV, Kielmann T (2005) MPJ/Ibis: a flexible and efficient message-passing platform for Java. In: Proc 12th EuroPVM/MPI (EuroPVM/MPI'05), LNCS, vol 3666, Sorrento, Italy, pp 217–224
11. Pugh B, Spacco J (2003) MPJava: High-performance message-passing in Java using Java.nio. In: Proc 16th int workshop on languages and compilers for parallel computing (LCPC'03), LNCS, vol 2958, College Station, TX, USA, pp 323–339
12. Taboada GL, Touriño J, Doallo R (2010) Performance analysis of message-passing libraries on high-speed clusters. Int J Comput Syst Sci Eng 25(1):63–78, January
13. 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
14. Barchet-Estefanel LA, Mounie G (2004) Fast tuning of intra-cluster collective communications. In: Proc 11th EuroPVM/MPI (EuroPVM/MPI'04), LNCS, vol 3241, Budapest, Hungary, pp 28–35
15. Pjesivac-Grbovic J, Angskun T, Bosilca G, Fagg GE, Gabriel E, Dongarra JJ (2007) Performance analysis of MPI collective operations. Cluster Comput 10(2):127–143
16. Thakur R, Rabenseifner R, Gropp W (2005) Optimization of collective communication operations in MPICH. Int J High Perform Comput Appl 19(1):49–66

17. Pjesivac-Grbovic J, Fagg GE, Angskun T, Bosilca G, Dongarra JJ (2006) MPI collective algorithm selection and quadtree encoding. In: 13th EuroPVM/MPI (EuroPVM/MPI'06), LNCS, vol 4192, Bonn, Germany, pp 40–48
18. Sanders P, Träff JL (2002) The hierarchical factor algorithm for all-to-all communication. In: Proc 8th int Euro-Par (Euro-Par'02), LNCS, vol 2400, Paderborn, Germany, pp 799–804
19. Zhu H, Goodell D, Gropp W, Thakur R (2009) Hierarchical collectives in MPICH2. In: Proc 16th EuroPVM/MPI (EuroPVM/MPI'09), LNCS, vol 5759, Espoo, Finland, pp 325–326
20. Tu B, Fan J, Zhan J, Zhao X (2010) Performance analysis and optimization of MPI collective operations on multi-core clusters. J Supercomp (in press)
21. Tipparaju V, Nieplocha J, Panda DK (2003) Fast collective operations using shared and remote memory access protocols on clusters. In: Proc 17th int parallel and distributed processing symposium (IPDPS'03), Nice, France, pp. 84–93
22. Mercier G, Clet-Ortega J (2009) Towards an efficient process placement policy for MPI applications in multicore environments. In: Proc 16th EuroPVM/MPI (EuroPVM/MPI'09), LNCS, vol 5759, Espoo, Finland, pp 104–115
23. Nelisse A, Maassen J, Kielmann T, Bal HE (2003) CCJ: object-based message-passing and collective communication in Java. Concurr Comput, Pract Exp 15(3–5):341–369
24. Lim S, Carpenter B, Fox G, Lee H (2005) Collective communications for scalable programming. In: Proc 3rd int symposium on parallel and distributed processing and applications (ISPA'05), LNCS, vol 3758, Nanjing, China, pp 286–297
25. Shafi A, Manzoor J (2009) Towards efficient shared memory communications in MPJ Express. In: Proc 11th int workshop on Java and components for parallelism, distribution and concurrency (IWJacPDC'09), Rome, Italy, p 111b (8 pages)
26. 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 int conf on cluster computing (CLUSTER'03), Hong Kong, China, pp 118–126
27. Mallón DA, Taboada GL, Touriño J, Doallo R (2009) NPB-MPJ: NAS parallel benchmarks implementation for message-passing in Java. In: Proc 17th euromicro int conf on parallel, distributed, and network-based processing (PDP'09), Weimar, Germany, pp 181–190
28. Baker M, Carpenter B, Shafi A (2006) MPJ Express meets Gadget: towards a Java code for cosmological simulations. In: 13th EuroPVM/MPI (EuroPVM/MPI'06), Bonn, Germany, pp 358–365
29. Finis Terrae. http://www.top500.org/system/9156 [Last visited: March 2010]
30. TOP500 supercomputing site. http://www.top500.org [Last visited: March 2010]