Java in the High Performance Computing Arena: Research, Practice and Experience

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

Computer Architecture Group
University of A Coruña, A Coruña (Spain)
{taboada,sramos,rreye,juan,doallo}@udc.es

Abstract

The rising interest in Java for High Performance Computing (HPC) is based on the appealing features of this language for programming multi-core cluster architectures, particularly the built-in networking and multithreading support, and the continuous increase in Java Virtual Machine (JVM) performance. However, its adoption in this area is being delayed by the lack of analysis of the existing programming options in Java for HPC and thorough and up-to-date evaluations of their performance, as well as the unawareness of the current research projects in this field, whose solutions are needed in order to boost the embracement of Java in HPC.

This paper analyzes the current state of Java for HPC, both for shared and distributed memory programming, presents related research projects, and finally, evaluates the performance of current Java HPC solutions and research developments on two shared memory environments and two InfiniBand multi-core clusters. The main conclusions are that: (1) the significant interest in Java for HPC has led to the development of numerous projects, although usually quite modest, which may have prevented a higher development of Java in this field; (2) Java can achieve almost similar performance to natively compiled languages, both for sequential and parallel applications, being an alternative for HPC programming; and (3) the recent advances in the efficient support of Java communications on shared memory and low-latency networks are bridging the gap between Java and natively compiled applications in HPC. Thus, the good prospects of Java in this area are attracting the attention of both industry and academia, which can take significant advantage of Java adoption in HPC.

Keywords:
Java, High Performance Computing, Performance Evaluation, Multi-core Architectures, Message-passing, Threads, Cluster, InfiniBand

1. Introduction

Java has become a leading programming language soon after its release, especially in web-based and distributed computing environments, and it is an emerging option for High Performance Computing (HPC) [1, 2]. The increasing interest in Java for parallel computing is based on its appealing characteristics: built-in networking and multithreading support, object orientation, platform independence, portability, type-safety, security, it has an extensive API and a wide community of developers, and finally, it is the main training language for computer science students. Moreover, performance is no longer an obstacle. The performance gap between Java and native languages (e.g., C and Fortran) 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. However, the use of Java in HPC is being delayed by the lack of analysis of the existing programming options in this area and thorough and up-to-date evaluations of their performance, as well as the unawareness of the current research projects in Java for HPC, whose solutions are needed in order to boost its adoption.

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 [3]. In this scenario Java represents an attractive choice for the

development of parallel applications as it is a multithreaded language and provides built-in networking 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) communication. Nevertheless, although the performance gap between Java and native languages is usually small for sequential applications, it can be particularly high for parallel applications when depending on inefficient communication libraries, which has hindered Java adoption for HPC. Therefore, current research efforts are focused on providing scalable Java communication middleware, especially on high-speed networks commonly used in HPC systems, such as InfiniBand or Myrinet.

The remainder of this paper is organized as follows. Section 2 analyzes the existing programming options in Java for HPC. Section 3 describes current research efforts in this area, with special emphasis on providing scalable communication middleware for HPC. A comprehensive performance evaluation of representative solutions in Java for HPC is presented in Section 4. Finally, Section 5 summarizes our concluding remarks.

2. Java for High Performance Computing

This section analyzes the existing programming options in Java for HPC, which can be classified into: (1) shared memory programming; (2) Java sockets; (3) Remote Method Invocation (RMI); and (4) Message-passing in Java. These programming options allow the development of both high level libraries and Java parallel applications.

2.1. Java Shared Memory Programming

There are several options for shared memory programming in Java for HPC, such as the use of Java threads, OpenMP-like implementations, and Titanium.

As Java has built-in multithreading support, the use of Java threads for parallel programming is quite extended due to its high performance, although it is a rather low-level option for HPC (work parallelization and shared data access synchronization are usually hard to implement). Moreover, this option is limited to shared memory systems, which provide less scalability than distributed memory machines. Nevertheless, its combination with distributed memory programming models can overcome this restriction. Finally, in order to partially relieve programmers from the low-level details of threads programming, Java has incorporated from the 1.5 specification the concurrency utilities, such as thread pools, tasks, blocking queues, and low-level high-performance primitives for advanced concurrent programming like CyclicBarrier.

The project Parallel Java (PJ) [4] has implemented several high level abstractions over these concurrency utilities, such as ParallelRegion (code to be executed in parallel), ParallelTeam (group of threads that execute a ParallelRegion) and ParallelForLoop (work parallelization among threads), allowing an easy thread-base shared memory programming. Moreover, PJ also implements the message-passing paradigm as it is intended for programming hybrid shared/distributed memory systems such as multi-core clusters.

There are two main OpenMP-like implementations in Java, JOMP [5] and JaMP [6]. JOMP consists of a compiler (written in Java, and built using the JavaCC tool) and a runtime library. The compiler translates Java source code with OpenMP-like directives to Java source code with calls to the runtime library, which in turn uses Java threads to implement parallelism. The whole system is "pure" Java (100% Java), and thus can be run on any JVM. Although the development of this implementation stopped in 2000, it has been used recently to provide nested parallelism on multi-core HPC systems [7]. Nevertheless, JOMP had to be optimized with some of the utilities of the concurrency framework, such as the replacement of the busy-wait implementation of the JOMP barrier by the more efficient java.util.concurrent.CyclicBarrier. The experimental evaluation of the hybrid Java message-passing + JOMP configuration (being the message-passing library thread-safe) showed up to 3 times higher performance than the equivalent pure message-passing scenario. Although JOMP scalability is limited to shared memory systems, its combination with distributed memory communication libraries (e.g., message-passing libraries) can overcome this issue. JaMP is the Java OpenMP-like implementation for Jackal [8], a software-based Java Distributed Shared Memory (DSM) implementation. Thus, this project is limited to this environment. JaMP has followed the JOMP approach, but taking advantage of the concurrency utilities, such as tasks, as it is a more recent project.

The OpenMP-like approach has several advantages over the use of Java threads, such as the higher level programming model with a code much closer to the sequential version and the exploitation of the familiarity with OpenMP,

thus increasing programmability. However, current OpenMP-like implementations are still preliminary works and lack efficiency (busy-wait JOMP barrier) and portability (JaMP).

Titanium [9] is an explicitly parallel dialect of Java developed at UC Berkeley which provides the Partitioned Global Address Space (PGAS) programming model, like UPC and Co-array Fortran, thus achieving higher programmability. Besides the features of Java, Titanium adds flexible and efficient multi-dimensional arrays and an explicitly parallel SPMD control model with lightweight synchronization. Moreover, it has been reported that it outperforms Fortran MPI code [10], thanks to its source-to-source compilation to C code and the use of native libraries, such as numerical and high-speed network communication libraries. However, Titanium presents several limitations, such as the avoidance of the use of Java threads and the lack of portability as it relies on Titanium and C compilers.

2.2. Java Sockets

Sockets are a low-level programming interface for network communication, which allows sending streams of data between applications. The socket API is widely extended and can be considered the standard low-level communication layer as there are socket implementations on almost every network protocol. Thus, sockets have been the choice for implementing in Java the lowest level of network communication. However, Java sockets usually lack efficient high-speed networks support [11], so it has to resort to inefficient TCP/IP emulations for full networking support. Examples of TCP/IP emulations are IP over InfiniBand (IPoIB), IPoMX on top of the Myrinet low-level library MX (Myrinet eXpress), and SCIP on SCI.

Java has two main sockets implementations, the widely extended Java IO sockets, and Java NIO (New I/O) sockets which provide scalable non-blocking communication support. However, both implementations do not provide high-speed network support nor HPC tailoring. Ibis sockets partly solve these issues adding Myrinet support and being the base of Ibis [12], a parallel and distributed Java computing framework. However, their implementation on top of the JVM sockets library limits their performance benefits.

Java Fast Sockets (JFS) [11] is our high performance Java socket implementation for HPC. As JVM IO/NIO sockets do not provide high-speed network support nor HPC tailoring, JFS overcomes these constraints by: (1) reimplementing the protocol for boosting shared memory (intra-node) communication; (2) supporting high performance native sockets communication over SCI Sockets, Sockets-MX, and Socket Direct Protocol (SDP), on SCI, Myrinet and InfiniBand, respectively; (3) avoiding the need of primitive data type array serialization; and (4) reducing buffering and unnecessary copies. Thus, JFS is able to reduce significantly JVM sockets communication overhead. Furthermore, its interoperability and user and application transparency through reflection allow for its immediate applicability on a wide range of parallel and distributed target applications.

2.3. Java Remote Method Invocation

The Java Remote Method Invocation (RMI) protocol allows an object running in one JVM to invoke methods on an object running in another JVM, providing Java with remote communication between programs equivalent to Remote Procedure Calls (RPCs). The main advantage of this approach is its simplicity, although the main drawback is the poor performance shown by the RMI protocol.

ProActive [13] is an RMI-based middleware for parallel, multithreaded and distributed computing focused on Grid applications. ProActive is a fully portable "pure" Java (100% Java) middleware whose programming model is based on a Meta-Object protocol. With a reduced set of simple primitives, this middleware simplifies the programming of Grid computing applications: distributed on Local Area Network (LAN), on clusters of workstations, or for the Grid. Moreover, ProActive supports fault-tolerance, load-balancing, mobility, and security. Nevertheless, the use of RMI as its default transport layer adds significant overhead to the operation of this middleware.

The optimization of the RMI protocol has been the goal of several projects, such as KaRMI [14], RMIX [15], Manta [16], Ibis RMI [12], and Opt RMI [17]. However, the use of non-standard APIs, the lack of portability, and the insufficient overhead reductions, still significantly larger than socket latencies, have restricted their applicability. Therefore, although Java communication middleware (e.g., message-passing libraries) used to be based on RMI, current Java communication libraries use sockets due to their lower overhead. In this case, the higher programming effort required by the lower-level API allows for higher throughput, key in HPC.

2.4. 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.

Soon after the introduction of Java, there have been several implementations of Java message-passing libraries (eleven projects are cited in [18]). However, most of them have developed their own MPI-like binding for the Java language. The two main proposed APIs are the mpiJava 1.2 API [19], which tries to adhere to the MPI C++ interface defined in the MPI standard version 2.0, but restricted to the support of the MPI 1.1 subset, and the JGF MPJ (Message-Passing interface for Java) API [20], which is the proposal of the Java Grande Forum (JGF) [21] to standardize the MPI-like Java API. The main differences among these two APIs lie on naming conventions of variables and methods.

The Message-passing in Java (MPJ) libraries can be implemented: (1) using Java RMI; (2) wrapping an underlying native messaging library like MPI through Java Native Interface (JNI); or (3) using Java sockets. Each solution fits with specific situations, but presents associated trade-offs. The use of Java RMI, a "pure" Java (100% Java) approach, as base for MPJ libraries, ensures portability, but it might not be the most efficient solution, especially in the presence of high speed communication 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.

The mpiJava library [22] consists of a collection of wrapper classes that call a native MPI implementation (e.g., MPICH2 or OpenMPI) through JNI. This wrapper-based approach provides efficient communication relying on native libraries, adding a reduced JNI overhead. 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 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 maintenance has been superseded by the development of MPJ Express [7], 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 shared memory (smpdev device) and 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).

Currently, MPJ Express is the most active projects in terms of uptake by the HPC community, presence on academia and production environments, and available documentation. This project is also stable and publicly available along with its source code.

In order to update the compilation of Java message-passing implementations presented in [18], this paper presents the projects developed since 2003, in chronological order:

- MPJava [23] is the first Java message-passing library implemented on Java NIO sockets, taking advantage of their scalability and high performance communications.
- Jcluster [24] is a message-passing library which provides both PVM-like and MPI-like APIs and is focused on automatic task load balance across large-scale heterogeneous clusters. However, its communications are based on UDP and it lacks high-speed networks support.
- Parallel Java (PJ) [4] is a "pure" Java parallel programming middleware that supports both shared memory programming (see Section 2.1) and an MPI-like message-passing paradigm, allowing applications to take advantage of hybrid shared/distributed memory architectures. However, the use of its own API makes its adoption difficult.
- P2P-MPI [25] is a peer-to-peer framework for the execution of MPJ applications on the Grid. Among its features are: (1) self-configuration of peers (through JXTA peer-to-peer technology); (2) fault-tolerance, based on process replication; (3) a data management protocol for file transfers on the Grid; and (4) an MPJ

implementation that can use either Java NIO or Java IO sockets for communications, although it lacks high-speed networks support. In fact, this project is tailored to grid computing systems, disregarding the performance aspects.

- MPJ/Ibis [26] is the only JGF MPJ API implementation up to now. This library can use either "pure" Java communications, or native communications on Myrinet. Moreover, there are two low-level communication devices available in Ibis for MPJ/Ibis communications: 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, and its Myrinet support is based on the GM library, which shows poorer performance than the MX library.

- JMPI [27] is an implementation which can use either Java RMI or Java sockets for communications. However, the reported performance is quite low (it only scales up to two nodes).

- Fast MPJ (F-MPJ) [28] is our Java message-passing implementation which provides high-speed networks support, both direct and through Java Fast Sockets (see Section 3.1). F-MPJ implements the mpiJava 1.2 API, the most widely extended, and includes a scalable MPJ collectives library [29].

Table 1 serves as a summary of the Java message-passing projects discussed in this section.

Table 1: Java message-passing projects overview

	Pure Java Impl.	Socket impl.		High-speed network support			API		
		Java IO	Java NIO	Myrinet	InfiniBand	SCI	mpiJava 1.2	JGF MPJ	Other APIs
MPJava [23]	✓		✓						✓
Jcluster [24]	✓	✓							✓
Parallel Java [4]	✓	✓							✓
mpiJava [22]				✓	✓	✓	✓		
P2P-MPI [25]	✓	✓	✓				✓		
MPJ Express [7]	✓		✓	✓			✓		
MPJ/Ibis [26]	✓	✓		✓				✓	
JMPI [27]	✓	✓							✓
F-MPJ [28]	✓	✓		✓	✓	✓	✓		

3. Java for HPC: Current Research

This section describes current research efforts in Java for HPC, which can be classified into: (1) design and implementation of low-level Java message-passing devices; (2) improvement of the scalability of Java message-passing collective primitives; (3) automatic selection of MPJ collective algorithms; (4) implementation and evaluation of efficient MPJ benchmarks; (5) language extensions in Java for parallel programming paradigms; and (6) Java libraries to support data parallelism. These ongoing projects are providing Java with several evaluations of their suitability for HPC, as well as solutions for increasing their performance and scalability in HPC systems with high-speed networks and hardware accelerators such as Graphics Processing Units (GPUs).

3.1. Low-level Java Message-passing Communication Devices

The use of pluggable low-level communication devices for high performance communication support is widely extended in native message-passing libraries. Both MPICH2 and OpenMPI include several devices on Myrinet, InfiniBand and shared memory. Regarding MPJ libraries, in MPJ Express the low-level xdev layer [7] provides communication devices for different interconnection technologies. The three implementations of the xdev API currently available are niodev (over Java NIO sockets), mxdev (over Myrinet MX), and smpdev (shared memory communication), which has been introduced recently [30]. This latter communication device has two implementations, one thread-based (pure Java) and the other based on native IPC resources.

F-MPJ communication devices conform with the xxdev API [28], which supports the direct communication of any serializable object without data buffering, whereas xdev, the API that xxdev is extending, does not support this direct communication, relying on a buffering layer (mpjbuf layer). Additional benefits of the use of this API are its flexibility, portability and modularity thanks to its encapsulated design.

The xxdev API (see Listing 1) has been designed with the goal of being simple and small, providing only basic communication methods in order to ease the development of xxdev devices. In fact, this API is composed of 13 simple methods, which implement basic message-passing operations, such as 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). Moreover, synchronous communications are also embraced (ssend and issend). However, these communication methods use ProcessID objects instead of using ranks as arguments to send and receive primitives. In fact, the xxdev layer is focused on providing basic communication methods and it does not deal with high level message-passing abstractions such as groups and communicators. Therefore, a ProcessID object unequivocally identifies a device object.

Listing 1: API of the xxdev.Device class

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

  Request isend(Object message, ProcessID dstID, int tag, int context);
  Request irecv(Object message, ProcessID srcID, int tag, int context, Status status);
  void send(Object message, ProcessID  dstID, int tag, int context);
  Status recv(Objecct message, ProcessID srcID, int tag, int context);
  Request issend(Object message, ProcessID dstID, int tag, int context);
  void ssend(Object message, ProcessID srcID, int tag, int context);

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

Figure 1 presents an overview of the F-MPJ communication devices on shared memory and cluster networks. From top to bottom, the communication support of MPJ applications run with F-MPJ is implemented in the device layer. Current F-MPJ communication devices are implemented either on JVM threads (smpdev, a thread-based device), on sockets over the TCP/IP stack (iodev on Java IO sockets), or on native communication layers such as Myrinet eXpress (mxdev) and InfiniBand Verbs (IBV) (ibvdev), which are accessed through JNI.

The initial implementation of F-MPJ included only one communication device, iodev, implemented on top of Java IO sockets, which therefore can rely on top of JFS and hence obtain high performance on shared memory and Gigabit Ethernet, SCI, Myrinet, and InfiniBand networks. However, the use of sockets in a communication device, despite the high performance provided by JFS, still represents an important source of overhead in Java communications. Thus, F-MPJ is including the direct support of communications on high performance native communication layers, such as MX and IBV.

The mxdev device implements the xxdev API on MX, which runs natively on Myrinet and high-speed Ethernet networks, such as 10 Gigabit Ethernet, relying on MXoE (MX over Ethernet) stack. As MX already provides a low-

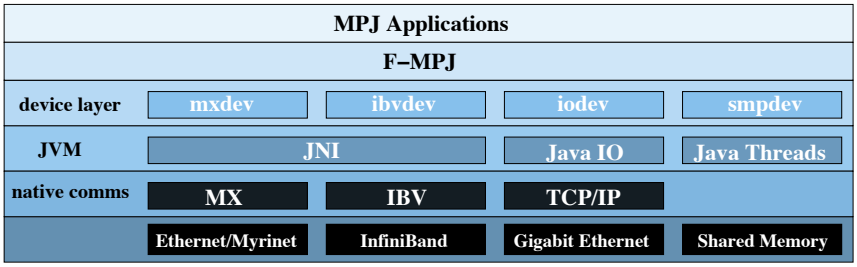

Figure 1: F-MPJ communication devices on shared memory and cluster networks

level messaging API, mxdev deals with the Java Objects marshaling and communication, the JNI transfers and the MX parameters handling. The ibvdev device implements the xxdev API on IBV, the low-level InfiniBand communication driver, in order to take full advantage of the InfiniBand network. Unlike mxdev, ibvdev has to implement its own communication protocols, as IBV API is quite close to the InfiniBand Network Interface Card (NIC) operation. Thus, this communication device has implemented two communication protocols, eager and rendezvous, on RDMA (Remote Direct Memory Access) Write/Send operations. This direct access of Java to InfiniBand network was somewhat restricted so far to MPI libraries. Like mxdev, this device has to deal with the Java Objects communication and the JNI transfers, and additionally with the communication protocols operation. Finally, both mxdev and ibvdev, although they have been primarily designed for network communication, support shared memory intra-node communication. However, smpdev device is the thread-based communication device that should support more efficiently shared memory transfers. This device isolates a naming space for each running thread (relying on custom class loaders) and allocates shared message queues in order to implementing the communications as regular data copies between threads.

3.2. MPJ Collectives Scalability

MPJ application developers use collective primitives for performing standard data movements (e.g., Broadcast, Scatter, Gather and Alltoall –total exchange–) 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, which generally consist of multiple point-to-point communications, 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.

The design, implementation and runtime selection of efficient collective communication operations have been extensively discussed in the context of native message-passing libraries [31, 32, 33, 34], while there is little discussion in MPJ, except for F-MPJ, which provides a scalable and efficient MPJ collective communication library [29] for parallel computing on multi-core architectures. This library provides multi-core aware primitives, implements several algorithms per collective operation, and explores thread-based communications, obtaining significant performance benefits in communication-intensive MPJ applications.

The collective algorithms present in MPJ libraries can be classified in six types, namely Flat Tree (FT) or linear, Minimum-Spanning Tree (MST), Binomial Tree (BT), Four-ary Tree (FaT), Bucket (BKT) or cyclic, and BiDirectional Exchange (BDE) or recursive doubling, which are extensively described in [32]. Table 2 presents a complete list of the collective algorithms used in MPJ Express and F-MPJ (the prefix "b" means that only blocking point-to-point communication is used, whereas "nb" refers to the use of non-blocking primitives). It can be seen that F-MPJ implements up to six algorithms per collective primitive, allowing their selection at runtime, as well as it takes more advantage of communications overlapping, achieving higher performance scalability. Regarding the memory requirements of the collective primitives, some algorithms require more memory than others (e.g., the MST algorithm for the Scatter and Gather demands more memory than the FT algorithm). Thus, when experiencing memory limitations the algorithms with less memory requirements must be selected in order to overcome the limitation.

Table 2: Algorithms implemented in MPJ collectives libraries.

Primitive	MPJ Express Collectives Library	F-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+Scatterv
Scan	nbFT	nbFT, linear

3.3. Automatic Selection of MPJ Collective Algorithms

The F-MPJ collectives library allows the runtime selection of the collective algorithm that provides the highest performance in a given multi-core system, among the several algorithms available, 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.

The information of the selected algorithms is stored in a configuration file that, if available in the system, is loaded at MPJ initialization, otherwise the default algorithms are selected, thus implementing a portable and user transparent approach.

The autotuning process consists of the execution of our own MPJ collectives micro-benchmark suite [18], 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 the performance measured by the micro-benchmark suite. Moreover, this autotuning process is required to be executed only once per system configuration in order to generate the configuration file. After that MPJ applications would take advantage of this information.

Table 3 presents the information contained in the optimum configuration file for the x86-64 multi-core cluster used in the experimental evaluation presented in this paper (Section 4). The thresholds between short and long messages, and between small and large number of processes are specific for each collective, although in the evaluated testbeds their values are generally 32 Kbytes and 16 processes, respectively.

3.4. Implementation and Evaluation of Efficient HPC Benchmarks

Java lacks efficient HPC benchmarking suites for characterizing its performance, although the development of efficient Java benchmarks and the assessment of their performance is highly important. The JGF benchmark suite [35], the most widely used Java HPC benchmarking suite, presents quite inefficient codes, as well as it does not provide the native language counterparts of the Java parallel codes, preventing their comparative evaluation. Therefore, we have implemented the NAS Parallel Benchmarks (NPB) suite for MPJ (NPB-MPJ) [36], selected as this suite is the most extended in HPC evaluations, with implementations for MPI (NPB-MPI), OpenMP (NPB-OMP), Java threads (NPB-JAV) and ProActive (NPB-PA).

NPB-MPJ allows, as main contributions: (1) the comparative evaluation of MPJ libraries; (2) the analysis of MPJ performance against other Java parallel approaches (e.g., Java threads); (3) the assessment of MPJ versus native MPI scalability; (4) the study of the impact on performance of the optimization techniques used in NPB-MPJ, from which

Table 3: Example of configuration file for the 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	linear	linear	linear	linear

Java HPC applications can potentially benefit. The description of the NPB-MPJ benchmarks implemented is next shown in Table 4.

Table 4: NPB-MPJ Benchmarks Description

Name	Operation	Communicat. intensiveness	Kernel	Applic.
CG	Conjugate Gradient	Medium	✓	
EP	Embarrassingly Parallel	Low	✓	
FT	Fourier Transformation	High	✓	
IS	Integer Sort	High	✓	
MG	Multi-Grid	High	✓	
SP	Scalar Pentadiagonal	Low		✓

In order to maximize NPB-MPJ performance, the "plain objects" design has been chosen as it reduces the overhead of the "pure" object-oriented design (up to 95% overhead reduction). Thus, each benchmark uses only one object instead of defining an object per each element of the problem domain. Thus, complex numbers are implemented as two-element arrays instead of complex numbers objects.

The inefficient multidimensional array support in Java (an n-dimensional array is defined as an array of $n-1$ dimensional arrays, so data is not guaranteed to be contiguous in memory) imposed a significant performance penalty in NPB-MPJ, which handle arrays of up to five dimensions. This overhead was reduced through the array flattening optimization, which consists of the mapping of a multidimensional array in a one-dimensional array. Thus, adjacent elements in the C/Fortran versions are also contiguous in Java, allowing the data locality exploitation.

Finally, the implementation of the NPB-MPJ takes advantage of the JVM JIT (Just-in-Time) compiler-based optimizations. The JIT compilation of the bytecode (or even its recompilation in order to apply further optimizations) is reserved to heavily-used methods, as it is an expensive operation that increases significantly the runtime. Thus, the NPB-MPJ codes have been refactored towards simpler and independent methods, such as methods for mapping elements from multidimensional to one-dimensional arrays, and complex number operations. As these methods are invoked more frequently, the JVM gathers more runtime information about them, allowing a more effective optimization of the target bytecode.

The performance of NPB-MPJ significantly improved using these techniques, achieving up to 2800% throughput increase (on SP benchmark). Furthermore, we believe that other Java HPC codes can potentially benefit from these optimization techniques.

3.5. Language Extensions in Java for Parallel Programming Paradigms

Regarding language extensions in Java to support various parallel programming paradigms, X10 and Habanero Java deserve to be mentioned. X10 [37, 38] is an emerging Java-based programming language developed in the

DARPA program on High Productivity Computer Systems (HPCS). Moreover, it is an APGAS (Asynchronous Partitioned Global Address Space) language implementation focused on programmability which supports locality exploitation, lightweight synchronization, and productive parallel programming. Additionally, an ongoing project based on X10 is Habanero Java [39], focused on supporting productive parallel programming on extreme scale homogeneous and heterogeneous multicore platforms. It allows to take advantage of X10 features in shared memory systems together with the Java Concurrency framework. Both X10 and Habanero Java applications can be compiled with C++ or Java backends, although looking for performance the use of the C++ one is recommended. Nevertheless, these are still experimental projects with limited performance, especially for X10 arrays handling, although X10 has been reported to rival Java threads performance on shared memory [40].

3.6. Java Libraries to Support Data Parallelism

There are several ongoing efforts in the support in Java of data parallelism using hardware accelerators, such as GPUs, once they have emerged as a viable alternative for significantly improving the performance of appropriate applications. On the one hand this support can be implemented in the compiler, at language level such as for JCUDA [41]. On the other hand, the interface to these accelerators can be library-based, such as the following Java bindings of CUDA: jcuda.org [42], jCUDA [43], JaCuda [44], Jacuzzi [45], and java-gpu [46].

Furthermore, the bindings are not restricted to CUDA as there are several Java bindings for OpenCL: jocl.org [47], JavaCL [48], and JogAmp [49].

This important number of projects is an example of the interest of the research community in supporting data parallelism in Java, although their efficiency is lower than using directly CUDA/OpenCL due to the overhead associated to the Java data movements to and from the GPU, the support of the execution of user-written CUDA code from Java programs and the automatic support for data transfer of primitives and multidimensional arrays of primitives. An additional project that targets these sources of inefficiency is JCudaMP [50], an OpenMP framework that exploits more efficiently GPUs. Finally, another approach for Java performance optimization on GPUs is the direct generation of GPU-executable code (without JNI access to CUDA/OpenCL) by a research Java compiler, Jikes, which is able to automatically parallelize loops [51].

4. Performance Evaluation

This paper presents an up-to-date comparative performance evaluation of representative MPJ libraries, F-MPJ and MPJ Express, two shared memory environments and two InfiniBand multi-core clusters. First, the performance of point-to-point MPJ primitives on InfiniBand, 10 Gigabit Ethernet and shared memory is presented. Next, this section evaluates the results gathered from a micro-benchmarking of MPJ collective primitives. Finally, the impact of MPJ libraries on the scalability of representative parallel codes, both NPB-MPJ kernels and the Gadget2 application [52], has been assessed comparatively with MPI, Java threads and OpenMP performance results.

4.1. Experimental Configuration

Two systems have been used in this performance evaluation, a multi-core x86-64 Infiniband cluster and the Finis Terrae supercomputer [53]. The first system (from now on x86-64 cluster) is a 16-node cluster with 16 Gbytes of memory and 2 x86-64 Xeon E5620 quad-core Nehalem-based "Gulftown" processors at 2.40 GHz per node (hence 128 physical cores in the cluster). The interconnection network is InfiniBand (QLogic IBA7220 4x DDR, 16 Gbps), although 2 of the nodes have additionally a 10 Gigabit Ethernet NIC (Intel PRO/10GbE NIC). As each node has 8 physical cores, and 16 logical cores when hyperthreading is enabled, shared memory performance has been also evaluated on one node of the cluster, using up to 16 processes/threads. The performance results on this system have been obtained using one core per node, except for 32, 64 and 128 processes, for which 2, 4 and 8 cores per node, respectively, have been used.

The OS is Linux CentOS 5.3, the C/Fortran compilers are the Intel compiler (used with -fast flag) version 11.1.073 and the GNU compiler (used with -O3 flag) version 4.1.2, both with OpenMP support, the native communication libraries are OFED (OpenFabrics Enterprise Distribution) 1.5 and Open-MX 1.3.4, for InfiniBand and 10 Gigabit Ethernet, respectively, and the JVM is Oracle JDK 1.6.0_23. Finally, the evaluated message-passing libraries are F-MPJ with JFS 0.3.1, MPJ Express 0.35, and OpenMPI 1.4.1.

The second system used is the Finis Terrae supercomputer (14 TFlops), an InfiniBand cluster which consists of 142 HP Integrity rx7640 nodes, each of them with 16 Montvale Itanium2 (IA64) cores at 1.6 GHz and 128 Gbytes of memory. The InfiniBand NIC is a 4X DDR Mellanox MT25208 (16 Gbps). Additionally an HP Integrity Superdome system with 64 Montvale Itanium 2 dual-core processors (total 128 cores) at 1.6 GHz and 1 TB of memory has also been used for the shared memory evaluation. The OS of the Finis Terrae is SUSE Linux Enterprise Server 10 with Intel compiler 10.1.074 (used with the -fast flag) and GNU compiler (used with the -O3 flag) version 4.1.2. Regarding native message-passing libraries, HP MPI 2.2.5.1 as been selected as it achieves the highest performance on InfiniBand and shared memory on the Finis Terrae. The InfiniBand drivers are OFED version 1.4. The JVM is Oracle JDK 1.6.0_20 for IA64. The poor performance of Java on IA64 architectures, due to the lack of mature support for this processor in the Java Just-In-Time compiler, has motivated the selection of this system only for the analysis of the performance scalability of MPJ applications, due to its high number of cores. The performance results on this system have been obtained using 8 cores per node, the recommended configuration for maximizing performance. In fact, the use of a higher number of cores per node increases significantly network contention and memory access bottlenecks.

Regarding the benchmarks, Intel MPI Benchmarks (IMB, formerly Pallas) and our own MPJ micro-benchmark suite, which tries to adhere to IMB measurement methodology, have been used for the message-passing primitives evaluation. Moreover, the NPB-MPI/NPB-OMP version 3.3 and the NPB-JAV version 3.0 have been used together with our own NPB-MPJ implementation [36]. The metrics that have been considered for the NPB evaluation are the speedup and MOPS (Millions of Operations Per Second), which measures the operations performed in the benchmark, that differ from the CPU operations issued. Moreover, NPB Class C workloads have been selected as they are the largest workloads that can be executed in a single node, which imposes the restriction of using workloads with memory requirements below 16 Gbytes (the amount of memory available in a node of the x86-64 cluster).

4.2. Performance Evaluation Methodology

All performance results presented in this paper are the median of 5 measurements in case of the kernels and applications and the median of up to the 1000 samples measured for the collective operations. The selection of the most appropriate performance evaluation methodology in Java has been thoroughly addressed in [54], concluding that the median is considered one of the best measures as its accuracy seems to improve with the number of measurements, which is in tune with the results reported in this paper.

Regarding the influence of JIT compilation in HPC performance results, the use of long-running codes (with runtimes of several hours and days) generally involves the use of a high percentage of JIT compiled code, which eventually improves performance. Moreover, the JVM execution mode selected for the performance evaluation is the default one (*mixed mode*) which compiles dynamically at runtime, based on profiling information, the bytecode of costly methods to native code, while interprets inexpensive pieces of code without incurring in runtime compilation overheads. Thus, this mode is able to provide higher performance than the use of the interpreted and even the compiled (an initial static compilation) execution modes. In fact, we have experimentally assessed the higher performance of the use of the mixed mode for the evaluated codes, whose percentage of runtime of natively compiled code is generally higher than 95% (hence, less than 5% of the runtime is generally devoted to interpreted code).

Furthermore, the non-determinism of JVM executions leads to oscillations in the time measures of Java applications. The main sources of variation are the JIT compilation and optimization in the JVM driven by a timer-based method sampling, thread scheduling, and garbage collection. However, the exclusive access to HPC resources and the characteristics of HPC applications (e.g., numerical intensive computation and a restricted use of object oriented features such as extensions and handling numerous objects) limit the variations in the experimental results of Java. In order to assess the variability of representative Java codes in HPC, the NPB kernels evaluated in this paper (CG, FT, IS and MG with Class C problem size) have been executed 40 times, both using F-MPJ and MPI, on 64 and 128 cores of the x86-64 cluster. Regarding message-passing primitives, both point-to-point and collectives include calls to native methods, which provide efficient communications on high-speed networks, thus obtaining performance results close to the theoretical limits of the network hardware. Moreover, their performance measures, when relying on native methods, provide results with little variation among iterations. Only message-passing transfers on shared memory present a high variability due to the scheduling of the threads on different cores within a node. In this scenario the performance results depend significantly on the scheduling of the threads on cores that belong to the same processor

and that even can share some cache levels. Nevertheless, due to space restrictions a detailed analysis of the impact of thread scheduling on Java communications performance can not be included in this paper. Thus, only the NPB kernels have been selected for the analysis of the performance variability of Java in HPC due to their balance in the combination of computation and communication as well as for their representativeness in HPC evaluation.

Figure 2 presents speedup graphs with box and whisker diagrams for the evaluated benchmarks, showing the measure of the minimum sample, the lower quartile (Q1), the median (Q2), upper quartile (Q3), and the maximum sample. The selected metric, speedup, has been selected for clarity purposes, as it allows a straightforward analysis of F-MPJ and MPI results, especially for the comparison of their range of values, which lie closer using speedups than other metrics such as execution times.

The analysis of the variability of the performance of these NPB kernels shows that F-MPJ results present similar variability as MPI codes, although for CG and FT on 128 cores the NPB-MPJ measures present higher variations than their natively compiled counterparts (MPI kernels). However, even in this scenario the variability of the Java codes is less than 10% of the speedup value (the measured speedups fall in the range of 90% and 110% of the median value), whereas the average variation is less than 5% of the speedup value. Furthermore, there is no clear evidence of the increase of the variability with the number of cores, except for NPB-MPJ CG and FT.

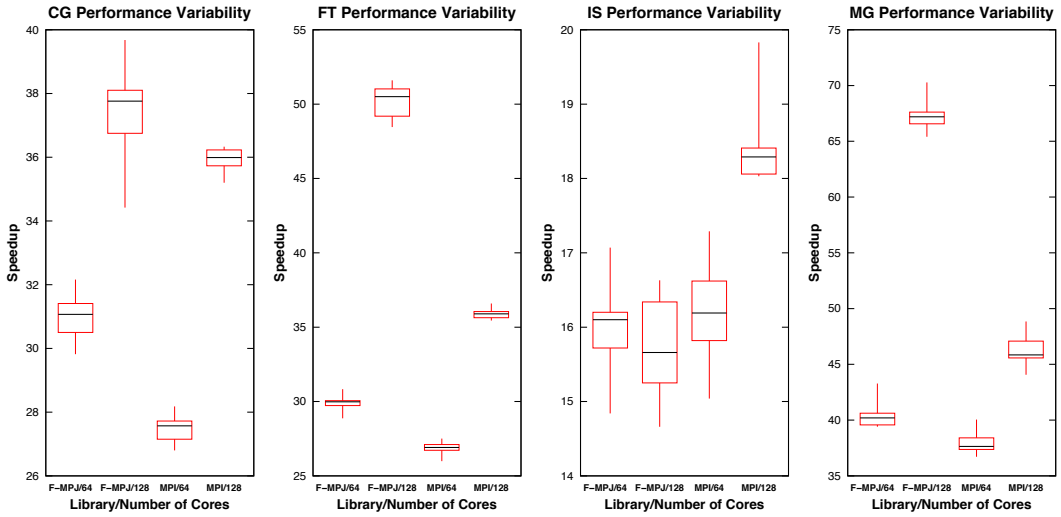

Figure 2: NPB performance variability on the x86-64 cluster

4.3. Experimental Performance Results on One Core

Figure 3 shows a performance comparison of several NPB implementations on one core from the x86-64 cluster (left graph) and on one core from the Finis Terrae (right graph). The results are shown in terms of speedup relative to the MPI library (using the GNU C/Fortran compiler), Runtime(*NPB-MPI benchmark*) / Runtime(*NPB benchmark*). Thus, a value higher than 1 means than the evaluated benchmark achieves higher performance (shorter runtime) than the NPB-MPI benchmark, whereas a value lower than 1 means than the evaluated code shows poorer performance (longer runtime) than the NPB-MPI benchmark. The NPB implementations and NPB kernels evaluated are those that will be next used in this section for the performance analysis of Java kernels (Section 4.6.1). Moreover, only F-MPJ results are shown for NPB-MPJ performance for clarity purposes, as other MPJ libraries (e.g., MPJ Express) obtain quite similar results on one core.

The differences in performance that can be noted in the graphs are explained by the different implementations of the NPB benchmarks, the use of Java or native code (C/Fortran), and for native code the compiler being used (Intel

or GNU compiler). Regarding Java performance, as the JVM used in this performance evaluation, the Oracle JVM for Linux, has been built with the GNU compiler, Java performance is limited by the throughput achieved with this compiler. Thus, Java codes (MPJ and Threads) cannot generally outperform their equivalent GNU-built benchmarks. This fact is of special relevance on the Finis Terrae, where the GNU compiler is not able to take advantage of the Montvale Itanium2 (IA64) processor, whereas the Intel compiler does. As a consequence of this, the performance of Java kernels on the Finis Terrae is significantly lower, even an order of magnitude lower, than the performance of the kernels built with the Intel compiler. The performance of Java kernels on the x86-64 cluster is close to the natively compiled kernels for CG and IS, whereas for FT and MG Java performance is approximately 55% of the performance of MPI kernels built with the GNU compiler.

This analysis of the performance of Java and natively compiled codes on the x86-64 cluster and the Finis Terrae has also verified that the use of the Intel compiler shows better performance results than the use of the GNU compiler, especially on the Finis Terrae. Thus, from now on only the Intel compiler has been used in the performance evaluation included in this paper, although a fair comparison with Java would have considered the GNU compiler (both Oracle JVM and the GNU compiler are freely available software). However, the use of the compiler provided by the processor vendor is the most generally adopted solution in HPC. Furthermore, a wider availability of JVMs built with commercial compilers would improve this scenario, especially on Itanium platforms.

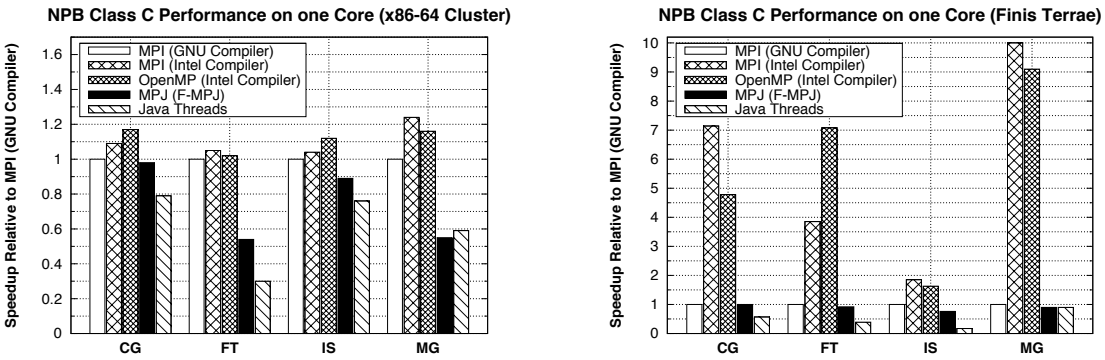

Figure 3: NPB relative performance on one core

4.4. Message-passing Point-to-point Micro-benchmarking

The performance of message-passing point-to-point primitives has been measured on the x86-64 cluster using our own MPJ micro-benchmark suite and IMB. Regarding Finis Terrae, its results are not considered for clarity purposes, as well as due to the poor performance of Java on this system. Moreover, Finis Terrae communication mechanisms, InfiniBand and shared memory, are already covered in the x86-64 cluster evaluation.

Figure 4 presents message-passing point-to-point latencies (for short messages) and bandwidths (for long messages) on InfiniBand (top graph), 10 Gigabit Ethernet (middle graph) and shared memory (bottom graph). Here, the results shown are the half of the round-trip time of a pingpong test or its corresponding bandwidth.

On the one hand these results show that F-MPJ is quite close to MPI performance, which means that F-MPJ is able to take advantage of the low latency and high throughput provided by shared memory and these high-speed networks. In fact, F-MPJ obtains start-up latencies as low as 2 μs on shared memory, 10 μs on InfiniBand and 12 μs on 10 Gigabit Ethernet. Regarding throughput, F-MPJ significantly outperforms MPI for 4 Kbytes and larger messages on shared memory when using smpdev communication device, achieving up to 51 Gbps thanks to the exploitation of the thread-based intra-process communication mechanism, whereas the inter-process communication protocols implemented in MPI and the F-MPJ network-based communication devices (ibvdev and mxdev) are limited to less than 31 Gbps.

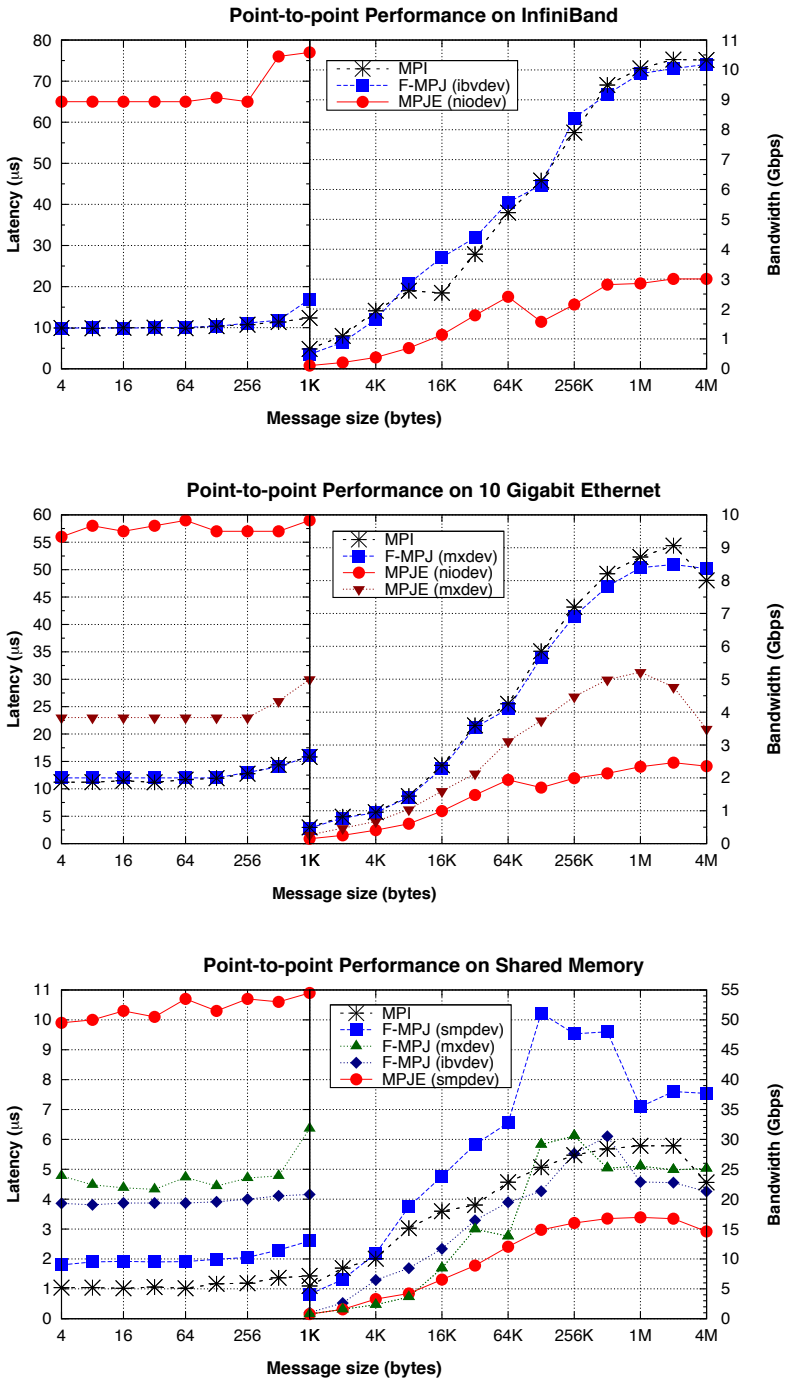

Figure 4: Message-passing point-to-point performance on InfiniBand, 10 Gigabit Ethernet and shared memory

On the other hand, MPJ Express point-to-point performance suffers from the lack of specialized support on InfiniBand, having to rely on NIO sockets over IP emulation IPoIB, and the use of a buffering layer, which adds noticeable overhead for long messages. Moreover, the communication protocols implemented in this library show a significant start-up latency. In fact, MPJ Express and F-MPJ rely on the same communication layer on shared memory (intra-process transfers) and 10 Gigabit Ethernet (Open-MX library), but MPJ Express adds an additional overhead of 8 μs and 11 μs, respectively, over F-MPJ.

4.5. Message-passing Collective Primitives Micro-benchmarking

Figure 5 presents the performance of representative message-passing data movement operations (Broadcast and Allgather), and computational operations (Reduce and Allreduce double precision sum operations), as well as their associated scalability using a representative message size (32 Kbytes). The results, obtained using 128 processes on the x86-64 cluster, are represented using aggregated bandwidth metric as this metric takes into account the global amount of data transferred, generally $message\ size * number\ of\ processes$.

The original MPJ Express collective primitives use the algorithms listed in Table 2 (column MPJ Express), whereas F-MPJ collectives library uses the algorithms that maximize the performance on this cluster according to the automatic performance tunning process. The selected algorithms are presented in Table 5, which extracts from the configuration file the most relevant information about the evaluated primitives.

The results confirm that F-MPJ is bridging the gap between MPJ and MPI collectives performance, but there is still room for improvement, especially when using several processes per node as F-MPJ collectives are not taking full advantage of the cores available within each node. The scalability graphs (right graphs) confirm this analysis, especially for the Broadcast and the Reduce operations.

Table 5: Algorithms that maximize performance on the x86-64 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

4.6. Java HPC Kernel/Application Performance Analysis

The scalability of Java for HPC has been analyzed using the NAS Parallel Benchmarks (NPB) implementation for MPJ (NPB-MPJ) [36]. The selection of the NPB has been motivated by its widespread adoption in the evaluation of languages, libraries and middleware for HPC. In fact, there are implementations of this benchmarking suite for MPI (NPB-MPI), Java Threads (NPB-JAV), OpenMP (NPB-OMP) and hybrid MPI/OpenMP (NPB-MZ). Four representative NPB codes, those with medium/high communication intensiveness (see Table 4), have been evaluated: CG (Conjugate Gradient), FT (Fourier Transform), IS (Integer Sort) and MG (Multi-Grid). Furthermore, the jGadget [55] cosmology simulation application has also been analyzed.

These MPJ codes have been selected for showing poor scalability in the related literature [1, 52]. Hence, these are target codes for the analysis of the scalability of current MPJ libraries, which have been evaluated using up to 128 processes on the x86-64 cluster, and up to 256 processes on the Finis Terrae.

4.6.1. Java NAS Parallel Benchmarks Performance Analysis

Figures 6 and 7 present the NPB CG, IS, FT and MG kernel results on the x86-64 cluster and Finis Terrae, respectively, for the Class C workload in terms of MOPS (Millions of Operations Per Second) (left graphs) and their corresponding scalability, in terms of speedup (right graphs). These four kernels (CG, IS, FT and MG) have been selected as they present medium or high communication intensiveness (see Table 4). The two remaining kernels, EP

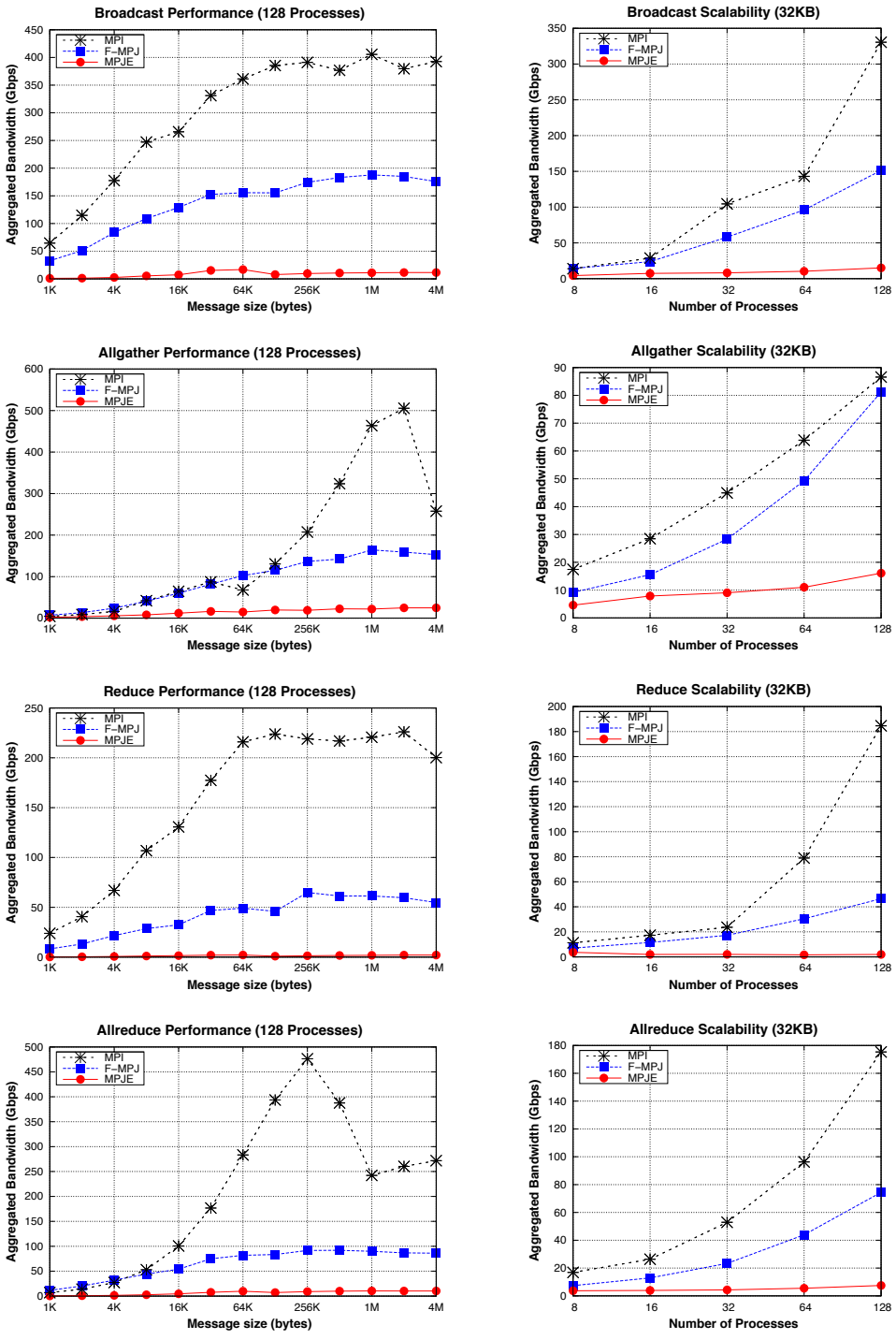

Figure 5: Collective primitives performance on the InfiniBand multi-core cluster

and SP, were discarded due to their low communication intensiveness (see Table 4) so their results show high scalability, having limited abilities to assess the impact of multithreading and MPJ libraries on the scalability of parallel codes. The NPB implementations used are NPB-MPI and NPB-MPJ for the message-passing scalability evaluation on distributed memory and NPB-OMP and NPB-JAV for the evaluation of shared memory performance.

Although the configuration of the shared and the distributed memory scenarios are different, they share essential features such as the processor and the architecture of the system, so their results are shown together in order to ease their comparison. Thus, Figure 6 presents NPB results of shared and distributed memory implementations measured in the x86-64 cluster. The selected NPB kernels (CG, IS, FT and MG) are implemented in the four NPB implementations evaluated, in fact the lack of some of these kernels has prevented the use of additional benchmark suites, such as the hybrid MPI/OpenMP NPB Multi-Zone (NPB-MZ), which does not implement any of these kernels.

NPB-MPI results have been obtained using the MPI library that achieves the highest performance on each system, OpenMPI on the x86-64 cluster and HP-MPI on the Finis Terrae, in both cases in combination with the Intel C/Fortran compiler. Regarding NPB-MPJ, both F-MPJ and MPJ Express have been benchmarked using the communication device that shows the best performance on InfiniBand, the interconnection network of both systems. Thus, F-MPJ has been run using its ibvdev device whereas MPJ Express relies on niodev over the IP emulation IPoIB. NPB-OMP benchmarks have been compiled with the OpenMP support included in the Intel C/Fortran compiler. Finally, NPB-JAV codes only require a standard JVM for running.

The analysis of the x86-64 cluster results (Figure 6) first reveals that F-MPJ achieves similar performance to OpenMPI for CG when using 32 and higher number of cores, showing higher speedups than the MPI library in this case. As this kernel only includes point-to-point communication primitives, F-MPJ takes advantage of obtaining similar point-to-point performance to MPI. However, MPJ Express and the Java threads implementations present poor scalability from 8 cores. On the one hand, the poor speedups of MPJ Express are direct consequence of the use of sockets and IPoIB in its communication layer. On the other hand, the poor performance of the NPB-JAV kernels is motivated by their inefficient implementation. In fact, the evaluated codes obtain lower performance on a single core than the MPI, OpenMP and MPJ kernels, except for NPB-JAV MG, which outperforms NPB-MPJ MG (see in Subsection 4.3 the left graph in Fig. 3). The reduced performance of NPB-JAV kernels on a single core, which can incur up to 50% performance overhead compared to NPB-MPJ codes, determines the lower overall performance in terms of MOPS.

Additionally, the NPB shared memory implementations, using OpenMP and Java Threads, present poorer scalability on the x86_64 cluster than distributed memory (message-passing) implementations, except for NPB-OMP IS. The main reason behind this behavior is the memory access overhead when running 8 and even 16 threads on 8 physical cores, which thanks to hyperthreading are able to run up to 16 threads simultaneously. Thus, the main performance bottleneck for these shared memory implementations is the access to memory, which limits their scalability and prevents taking advantage of enabling hyperthreading.

Regarding FT results, although F-MPJ scalability is higher than MPI (F-MPJ speedup is about 50 on 128 cores whereas the MPI one is below 36), this is not enough for achieving similar performance in terms of MOPS. In this case MPJ performance is limited by its poor performance on one core, which is 54% of the MPI performance (see in Subsection 4.3 the left graph in Fig. 3). Moreover, the scalability of this kernel relies on the performance of the Alltoall collective, which has not prevented F-MPJ scalability. As for CG, MPJ Express and the shared memory NPB codes show poor performance, although NPB-JAV FT presents a slightly performance benefit when resorting to hyperthreading, probably due to its poor performance on one core, which is below 30% of the NPB-MPI FT result. In fact, a longer runtime reduces the impact of communications and memory bottlenecks in the scalability of parallel codes.

The significant communication intensiveness of IS, the highest among the evaluated kernels, reduces the observed speedups, which are below 20 on 128 cores. On the one hand, the message-passing implementations of this kernel rely heavily on Alltoall and Allreduce primitives, whose overhead is the main performance penalty. In fact, F-MPJ scalability drops from 64 cores (MPJ Express from 32 cores), whereas MPI shows poor scalability from 64 cores (the performance comparison between 64 and 128 cores shows that the use of the additional 64 cores only increases the speedup in 3 units, from 16 to 19). On the other hand, OpenMP IS obtains the best results on 8 cores, showing a high parallel efficiency, and even takes advantage of the use of hyperthreading. However, the implementation of IS using Java threads shows very poor scalability, with speedups below 2.

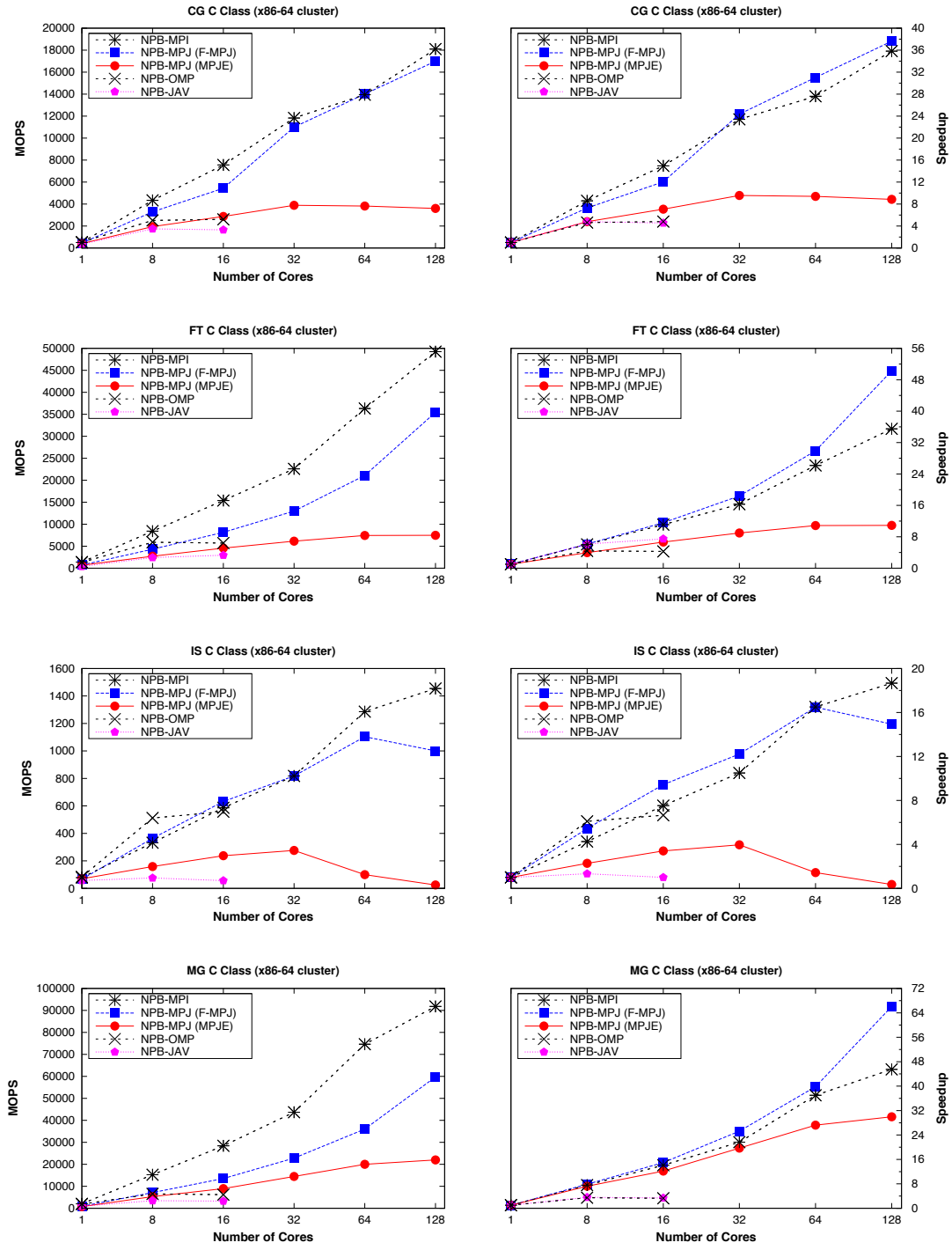

Figure 6: NPB Class C results on the x86-64 cluster

The highest MG performance in terms of MOPS has been obtained with MPI, followed at a significant distance by F-MPJ although this Java library shows higher speedups, especially on 128 cores. The reason, as for FT, is that MPJ performance is limited by its poor performance on one core, which is 55% of the MPI performance (see in Subsection 4.3 the left graph in Fig. 3). The longer MPJ runtime contributes to achieve high speedups in MG, trading off the bottleneck that represents the extensive use by this kernel of Allreduce, a collective whose performance is lower for MPJ than for MPI. In fact, the message-passing implementations of this kernel, both MPI and MPJ, present relatively good scalability, even for MPJ Express which achieves speedups around 30 on 64 and 128 cores. Nevertheless, the shared memory codes show little speedups, below 4 on 8 cores.

Figure 7 shows the Finis Terrae results, where the message-passing kernel implementations, NPB-MPI and NPB-MPJ, have been run on the rx7640 nodes of this supercomputer, using 8 cores per node and up to 32 nodes (hence up to 256 cores), whereas the shared memory results (NPB-OMP and NPB-JAV) have been obtained from the HP Integrity Superdome using up to 128 cores. Although the results have been obtained using two different hardware configurations, both subsystems share the same features but the memory architecture, which is distributed in rx7640 nodes and shared in the Integrity Superdome, as presented in Subsection 4.1.

The analysis of the Finis Terrae results (Figure 7) shows that the best performer is OpenMP, showing significantly higher MOPS than the other implementations, except for MG where it is outperformed by MPI. Nevertheless, OpenMP suffers scalability losses from 64 cores due to the access to remote cells and the relative poor bidirectional traffic performance in the cell controller (the Integrity Superdome is a ccNUMA system which consists of 16 cells, each one with 4 dual-core processors and 64 Gbytes memory, interconnected through a crossbar network) [56]. The high performance of OpenMP contrasts with the poor results in terms of MOPS of NPB-JAV, although this is motivated by its poor performance on one core, which is usually an order of magnitude lower than MPI (Intel Compiler) performance (see in Subsection 4.3 the right graph in Fig. 3). Although this poor runtime favors the obtaining of high scalability, in fact NPB-JAV obtains speedups above 30 for CG and FT, this is not enough to bridge the gap with OpenMP results as NPB-OMP codes achieves even higher speedups, except for FT. Furthermore, NPB-JAV results are significantly poorer than those of NPB-MPJ (around 2-3 times lower), except for MG, which confirms the inefficiency of this Java threads implementation.

The performance results of the message-passing codes, NPB-MPI and NPB-MPJ, are between NPB-OMP kernels and the shared memory implementations, except for NPB-MPI MG, which is the best performer for MG kernel. Nevertheless, there are significant differences among the libraries been used. Thus, MPJ Express presents modest speedups, below 30, due to the use of a sockets-based (niodev) communication device over the IP emulation IPoIB. This limitation is overcome in F-MPJ, relying more directly on IBV. Thus, F-MPJ is able to achieve the highest speedups, motivated in part by the longer runtimes on one core (see in Subsection 4.3 the right graph in Fig. 3) which favor this scalability (a heavy workload reduces the impact of communications on the overall performance scalability). The high speedups of F-MPJ, which are significantly higher than those of MPI (e.g., up to 7 times higher in CG), allow F-MPJ to bridge the gap between Java and natively compiled languages in HPC. In fact, F-MPJ performance results for CG and FT on 256 are close to those of MPI, although their performance on one core is around 7 and 4 times lower than MPI results for CG and FT, respectively.

The analysis of these NPB experimental results show that the performance of MPJ libraries heavily depends on their InfiniBand support. Thus, F-MPJ, which relies directly on IBV, outperforms significantly MPJ Express, whose socket-based communication device runs on IPoIB, obtaining relatively low performance, especially in terms of start-up latency. Furthermore, NPB-MPJ kernels have revealed to be the most efficient Java implementation, significantly outperforming Java threads implementations, both in terms of performance on one core and scalability. Moreover, the comparative evaluation of NPB-MPJ and NPB-MPI results reveals that efficient MPJ libraries can help to bridge the gap between Java and native code performance in HPC. Finally, the evaluated libraries have shown higher speedups on Finis Terrae than on the x86-64 cluster. The reason behind this behavior is that the obtaining of poorer performance on one core allows for higher scalability given the same interconnection technology (both systems use 16 Gbps InfiniBand DDR networks). Thus, NPB-MPJ kernels on the Finis Terrae, showing some of the poorest performance on one core, are able to achieve speedups of up to 175 on 256 cores, whereas NPB-MPI scalability on the x86-64 cluster is always below a speedup of 50. Nevertheless, NPB-MPI on the x86-64 cluster shows the highest performance in terms of MOPS, outperforming NPB-MPI results on the Finis Terrae, which has double the number of available cores (256 cores available on the Finis Terrae vs. 128 cores available on the x86-64 cluster).

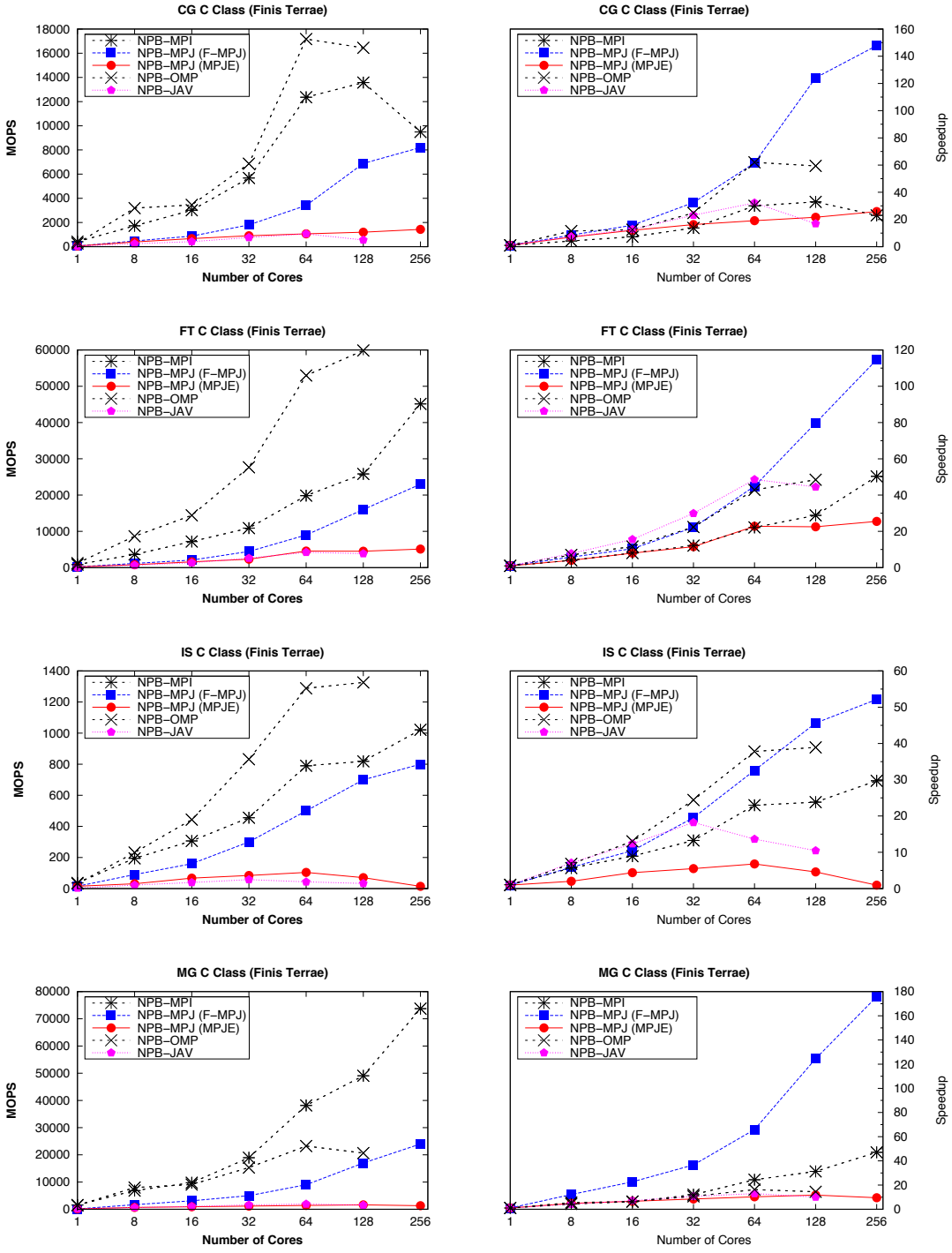

Figure 7: NPB Class C results on Finis Terrae

4.6.2. Performance Analysis of the jGadget Application

The jGadget [55] application is the MPJ implementation of Gadget [57], a popular cosmology simulation code initially implemented in C and parallelized using MPI that is used to study a large variety of problems like colliding and merging galaxies or the formation of large-scale structures. The parallelization strategy, both with MPI and MPJ, is an irregular and dynamically adjusted domain decomposition, with copious communication between processes. jGadget has been selected as representative Java HPC application as its performance has been previously analyzed [52] for their Java (MPJ) and C (MPI) implementations, as well as for its communication intensiveness and its popularity.

Figure 8 presents jGadget and Gadget performance results on the x86-64 cluster and the Finis Terrae for a galaxy cluster formation simulation with 2 million particles in the system (simulation available within the examples of Gadget software bundle). As Gadget is a communication-intensive application, with significant collective operations overhead, its scalability is modest, obtaining speedups of up to 48 on 128 cores of the x86-64 cluster and speedups of up to 57 on 256 cores of the Finis Terrae. Here F-MPJ achieves generally the highest speedups, followed closely by MPI, except from 64 cores on the Finis Terrae where MPI loses performance. This slowdown is shared with MPJ Express, which shows its highest performance on 64 cores for both systems. Nevertheless, MPJ Express speedups on the Finis Terrae are much higher (up to 37) than on the x86-64 cluster (only up to 16), something motivated by the different runtime of the application on the x86-cluster and the Finis Terrae. In fact, MPI Gadget presents numerous library dependencies, such as FFTW-MPI, Hierarchical Data Format (HDF) support, and the numerical GNU Scientific Library (GSL), which are not fully optimized for this system, thus increasing significantly its runtime. An example of inefficiency is that GSL shows poor performance on the Finis Terrae. Here the use of Intel Math Kernel Library (MKL) would show higher performance but the support for this numerical library is not implemented in Gadget. As a consequence of this jGadget performs better, compared in relative terms with MPI, on the Finis Terrae (only 2 times slower than MPI) than on the x86-64 cluster (3 times slower than MPI), although the performance of Java on IA64 architectures is quite poor.

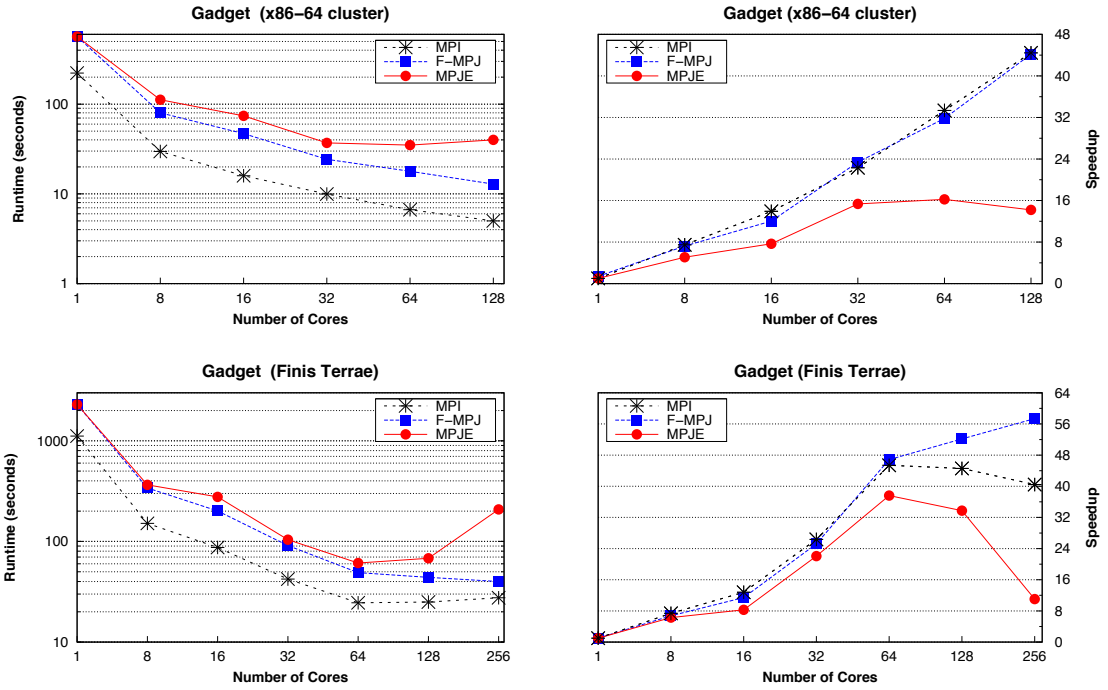

Figure 8: Gadget runtime and scalability on the x86-64 cluster and the Finis Terrae supercomputer

Moreover, the performance gap between Gadget and jGadget is motivated by the poor performance of the numerical methods included in jGadget, which consist of a translation of the GSL functions invoked in the Gadget source code, without relying on external numerical libraries. The use of an efficient Java numerical library [58], comparable in performance to Fortran numerical codes, would have improved the performance of jGadget. The development of such a library is still an ongoing effort, although it started a decade ago when it was demonstrated that Java was able to compete with Fortran in high performance numerical computing [59, 60]. In the last years a few projects are being actively developed [61], such as Universal Java Matrix Package (UJMP) [62], Efficient Java Matrix Library (EJML) [63], Matrix Toolkit Java (MTJ) [64] and jblas [65], which are replacing more traditional frameworks such as JAMA [66]. Furthermore, a recent evaluation of Java for numerical computing [67] has shown that the performance of Java applications can be significantly enhanced by delegating numerically intensive tasks to native libraries (e.g., Intel Math Kernel Library –MKL–) which supports the development of efficient high performance numerical applications in Java.

5. Conclusions

This paper has analyzed the current state of Java for HPC, both for shared and distributed memory programming, showing an important number of past and present projects which are the result of the sustained interest in the use of Java for HPC. Nevertheless, most of these projects are restricted to experimental environments, which prevents its general adoption in this field. However, the analysis of the existing programming options and available libraries in Java for HPC, together with the presentation in this paper of our current research efforts in the improvement of the scalability of our Java message-passing library, F-MPJ, would definitively contribute to boost the embracement of Java in HPC.

Additionally, Java lacks thorough and up-to-date evaluations of their performance in HPC. In order to overcome this issue this paper presents the performance evaluation of current Java HPC solutions and research developments on two shared memory environments and two InfiniBand multi-core clusters. The main conclusions of the analysis of these results is that Java can achieve almost similar performance to natively compiled languages, both for sequential and parallel applications, being an alternative for HPC programming. In fact, the performance overhead that Java may impose is a reasonable trade-off for the appealing features that this language provides for parallel programming multi-core architectures. Furthermore, the recent advances in the efficient support of Java communications on shared memory and low-latency networks are bridging the performance gap between Java and more traditional HPC languages.

Finally, the active research efforts in this area are expected to bring in the next future new developments that will continue rising the interest of both industry and academia and increasing the benefits of the adoption of Java for HPC.

Acknowledgments

This work was funded by the Ministry of Science and Innovation of Spain under Project TIN2010-16735 and an FPU grant AP2009-2112. We gratefully thank CESGA (Galicia Supercomputing Center, Santiago de Compostela, Spain) for providing access to the Finis Terrae supercomputer.

References

[1] G. L. Taboada, J. Touriño, R. Doallo, Java for High Performance Computing: Assessment of Current Research and Practice, in: Proc. 7th International Conference on the Principles and Practice of Programming in Java (PPPJ'09), Calgary, Alberta, Canada, 2009, pp. 30–39.

[2] B. Amedro, D. Caromel, F. Huet, V. Bodnartchouk, C. Delb, G. L. Taboada, ProActive: Using a Java Middleware for HPC Design, Implementation and Benchmarks, International Journal of Computers and Communications 3 (3) (2009) 49–57.

[3] J. Dongarra, D. Gannon, G. Fox, K. Kennedy, The Impact of Multicore on Computational Science Software, CTWatch Quarterly 3 (1) (2007) 1–10.

[4] A. Kaminsky, 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 (IWJacPDC'07), Long Beach, CA, USA, 2007, p. 196a (8 pages).

[5] M. E. Kambites, J. Obdrzálek, J. M. Bull, An OpenMP-like Interface for Parallel Programming in Java, Concurrency and Computation: Practice and Experience 13 (8-9) (2001) 793–814.

[6] M. Klemm, M. Bezold, R. Veldema, M. Philippsen, JaMP: an Implementation of OpenMP for a Java DSM, Concurrency and Computation: Practice and Experience 19 (18) (2007) 2333–2352.

[7] A. Shafi, B. Carpenter, M. Baker, Nested Parallelism for Multi-core HPC Systems using Java, Journal of Parallel and Distributed Computing 69 (6) (2009) 532–545.

[8] R. Veldema, R. F. H. Hofman, R. Bhoedjang, H. E. Bal, Run-time Optimizations for a Java DSM Implementation, Concurrency and Computation: Practice and Experience 15 (3-5) (2003) 299–316.

[9] K. A. Yelick, et al., Titanium: A High-performance Java Dialect, Concurrency - Practice and Experience 10 (11-13) (1998) 825–836.

[10] K. Datta, D. Bonachea, K. A. Yelick, Titanium Performance and Potential: An NPB Experimental Study, in: Proc. 18th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC'05), LNCS vol. 4339, Hawthorne, NY, USA, 2005, pp. 200–214.

[11] G. L. Taboada, J. Touriño, R. Doallo, Java Fast Sockets: Enabling High-speed Java Communications on High Performance Clusters, Computer Communications 31 (17) (2008) 4049–4059.

[12] R. V. v. Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, H. E. Bal, Ibis: a Flexible and Efficient Java-based Grid Programming Environment, Concurrency and Computation: Practice and Experience 17 (7-8) (2005) 1079–1107.

[13] L. Baduel, F. Baude, D. Caromel, Object-oriented SPMD, in: Proc. 5th IEEE Intl. Symposium on Cluster Computing and the Grid (CCGrid'05), Cardiff, UK, 2005, pp. 824–831.

[14] M. Philippsen, B. Haumacher, C. Nester, More Efficient Serialization and RMI for Java, Concurrency: Practice and Experience 12 (7) (2000) 495–518.

[15] D. Kurzyniec, T. Wrzosek, V. Sunderam, A. Slominski, RMIX: A Multiprotocol RMI Framework for Java, in: Proc. 5th Intl. Workshop on Java for Parallel and Distributed Computing (IWJPDC'03), Nice, France, 2003, p. 140 (7 pages).

[16] J. Maassen, R. V. v. Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, R. Hofman, Efficient Java RMI for Parallel Programming, ACM Transactions on Programming Languages and Systems 23 (6) (2001) 747–775.

[17] G. L. Taboada, C. Teijeiro, J. Touriño, High Performance Java Remote Method Invocation for Parallel Computing on Clusters, in: Proc. 12th IEEE Symposium on Computers and Communications (ISCC'07), Aveiro, Portugal, 2007, pp. 233–239.

[18] G. L. Taboada, J. Touriño, R. Doallo, 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.

[19] B. Carpenter, G. Fox, S.-H. Ko, S. Lim, mpiJava 1.2: API Specification, http://www.hpjava.org/reports/mpiJava-spec/mpiJava-spec/mpiJava-spec.html [Last visited: May 2011].

[20] B. Carpenter, V. Getov, G. Judd, A. Skjellum, G. Fox, MPJ: MPI-like Message Passing for Java, Concurrency: Practice and Experience 12 (11) (2000) 1019–1038.

[21] Java Grande Forum, http://www.javagrande.org, [Last visited: May 2011].

[22] M. Baker, B. Carpenter, G. Fox, S. Ko, S. Lim, mpiJava: an Object-Oriented Java Interface to MPI, in: Proc. 1st Intl. Workshop on Java for Parallel and Distributed Computing (IWJPDC'99), LNCS vol. 1586, San Juan, Puerto Rico, 1999, pp. 748–762.

[23] B. Pugh, J. Spacco, MPJava: High-Performance Message Passing in Java using Java.nio, in: Proc. 16th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC'03), LNCS vol. 2958, College Station, TX, USA, 2003, pp. 323–339.

[24] B.-Y. Zhang, G.-W. Yang, W.-M. Zheng, Jcluster: an Efficient Java Parallel Environment on a Large-scale Heterogeneous Cluster, Concurrency and Computation: Practice and Experience 18 (12) (2006) 1541–1557.

[25] S. Genaud, C. Rattanapoka, P2P-MPI: A Peer-to-Peer Framework for Robust Execution of Message Passing Parallel Programs, Journal of Grid Computing 5 (1) (2007) 27–42.

[26] M. Bornemann, R. V. v. Nieuwpoort, T. Kielmann, MPJ/Ibis: a Flexible and Efficient Message Passing Platform for Java, in: Proc. 12th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'05), LNCS vol. 3666, Sorrento, Italy, 2005, pp. 217–224.

[27] S. Bang, J. Ahn, Implementation and Performance Evaluation of Socket and RMI based Java Message Passing Systems, in: Proc. 5th ACIS Intl. Conf. on Software Engineering Research, Management and Applications (SERA'07), Busan, Korea, 2007, pp. 153 – 159.

[28] G. L. Taboada, J. Touriño, R. Doallo, F-MPJ: Scalable Java Message-passing Communications on Parallel Systems, Journal of Supercomputing (In press).

[29] G. L. Taboada, S. Ramos, J. Touriño, R. Doallo, Design of Efficient Java Message-passing Collectives on Multi-core Clusters, Journal of Supercomputing 55 (2) (2011) 126–154.

[30] A. Shafi, J. Manzoor, K. Hameed, B. Carpenter, M. Baker, Multicore-enabling the MPJ Express Messaging Library, in: Proc. 8th International Conference on the Principles and Practice of Programming in Java (PPPJ'10), Vienna, Austria, 2010, pp. 49–58.

[31] L. A. Barchet-Estefanel, G. Mounie, Fast Tuning of Intra-cluster Collective Communications, in: Proc. 11th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'04), LNCS vol. 3241, Budapest, Hungary, 2004, pp. 28 – 35.

[32] E. Chan, M. Heimlich, A. Purkayastha, R. A. van de Geijn, Collective Communication: Theory, Practice, and Experience, Concurrency and Computation: Practice and Experience 19 (13) (2007) 1749–1783.

[33] R. Thakur, R. Rabenseifner, W. Gropp, Optimization of Collective Communication Operations in MPICH, Intl. Journal of High Performance Computing Applications 19 (1) (2005) 49–66.
[34] S. S. Vadhiyar, G. E. Fagg, J. J. Dongarra, Towards an Accurate Model for Collective Communications, Intl. Journal of High Performance Computing Applications 18 (1) (2004) 159–167.
[35] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, R. A. Davey, A Benchmark Suite for High Performance Java, Concurrency: Practice and Experience 12 (6) (2000) 375–388.
[36] D. A. Mallón, G. L. Taboada, J. Touriño, R. Doallo, NPB-MPJ: NAS Parallel Benchmarks Implementation for Message-Passing in Java, in: Proc. 17th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing (PDP'09), Weimar, Germany, 2009, pp. 181–190.
[37] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, V. Sarkar, X10: an Object-oriented Approach to non-uniform Cluster Computing, in: Proc. 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05), San Diego, CA, USA, 2005, pp. 519–538.
[38] X10: Performance and Productivity at Scale, http://x10plus.cloudaccess.net/ [Last visited: May 2011].
[39] Habanero Java, habanero.rice.edu/hj.html [Last visited: May 2011].
[40] J. Shirako, H. Kasahara, V. Sarkar, Language Extensions in Support of Compiler Parallelization, in: Proc. 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC'07), Urbana, IL, USA, 2007, pp. 78–94.
[41] Y. Yan, M. Grossman, V. Sarkar, JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA, in: Proc. 15th International European Conference on Parallel and Distributed Computing (Euro-Par'09), Delft, The Netherlands, 2009, pp. 887–899.
[42] jcuda.org, http://jcuda.org [Last visited: May 2011].
[43] jCUDA, http://hoopoe-cloud.com/Solutions/jCUDA/Default.aspx [Last visited: May 2011].
[44] JaCuda, http://jacuda.sourceforge.net [Last visited: May 2011].
[45] Jacuzzi, http://sourceforge.net/apps/wordpress/jacuzzi [Last visited: May 2011].
[46] java-gpu, http://code.google.com/p/java-gpu [Last visited: May 2011].
[47] jocl.org, http://jocl.org [Last visited: May 2011].
[48] JavaCL, http://code.google.com/p/javacl [Last visited: May 2011].
[49] JogAmp, http://jogamp.org [Last visited: May 2011].
[50] G. Dotzler, R. Veldema, M. Klemm, JCudaMP: OpenMP/Java on CUDA, in: Proc. 3rd International Workshop on Multicore Software Engineering (IWMSE'10), Cape Town, South Africa, 2010, pp. 10–17.
[51] A. Leung, O. Lhoták, G. Ghulam Lashari, Automatic Parallelization for Graphics Processing Units, in: Proc. 7th International Conference on the Principles and Practice of Programming in Java (PPPJ'09), Calgary, Alberta, Canada, 2009, pp. 91–100.
[52] A. Shafi, B. Carpenter, M. Baker, A. Hussain, A Comparative Study of Java and C Performance in two Large-scale Parallel Applications, Concurrency and Computation: Practice and Experience In press.
[53] Finis Terrae Supercomputer, Galicia Supercomputing Center (CESGA), http://www.top500.org/system/9156 [Last visited: May 2011].
[54] A. Georges, D. Buytaert, L. Eeckhout, Statistically Rigorous Java Performance Evaluation, in: Proc. 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07), Montreal, Quebec, Canada, 2007, pp. 57–76.
[55] M. Baker, B. Carpenter, A. Shafi, MPJ Express Meets Gadget: Towards a Java Code for Cosmological Simulations, in: Proc. 13th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'06), Bonn, Germany, 2006, pp. 358–365.
[56] D. Mallón, G. Taboada, C. Teijeiro, J. Touriño, B. Fraguela, A. Gómez, R. Doallo, J. Mouriño, Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures, in: Proc. 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'09), Espoo, Finland, 2009, pp. 174–184.
[57] V. Springel, The Cosmological Simulation Code GADGET-2, Monthly Notices of the Royal Astronomical Society 364 (4) (2005) 1105–1134.
[58] JavaGrande JavaNumerics, http://math.nist.gov/javanumerics/ [Last visited: May 2011].
[59] R. F. Boisvert, J. J. Dongarra, R. Pozo, K. A. Remington, G. W. Stewart, Developing Numerical Libraries in Java, Concurrency: Practice and Experience 10 (11-13) (1998) 1117–1129.
[60] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, R. D. Lawrence, Java Programming for High-Performance Numerical Computing, IBM Systems Journal 39 (1) (2000) 21–56.
[61] H. Arndt, M. Bundschus, A. Naegele, Towards a Next-Generation Matrix Library for Java, in: Proc. 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09), Seattle, WA, USA, 2009, pp. 460–467.
[62] Universal Java Matrix Package (UJMP), http://www.ujmp.org [Last visited: May 2011].
[63] Efficient Java Matrix Library (EJML), http://code.google.com/p/efficient-java-matrix-library/ [Last visited: May 2011].
[64] Matrix Toolkits Java (MTJ), http://code.google.com/p/matrix-toolkits-java/ [Last visited: May 2011].
[65] Linear Algebra for Java (jblas), http://jblas.org/ [Last visited: May 2011].
[66] JAMA: A Java Matrix Package, http://math.nist.gov/javanumerics/jama [Last visited: May 2011].
[67] M. Baitsch, N. Li, D. Hartmann, A Toolkit for Efficient Numerical Applications in Java, Advances in Engineering Software 41 (1) (2010) 75–83.