Java for High Performance Computing: Assessment of Current Research and Practice

Guillermo L. Taboada, Juan Touriño, Ramón Doallo
Computer Architecture Group
University of A Coruña, A Coruña (Spain)
{taboada,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 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, pre-- sents related research projects, and finally, evaluates the performance of current Java HPC solutions and research developments on a multi-core cluster with a high-speed network, InfiniBand, and a 24-core shared memory machine. The main conclusions are that: (1) the significant interest on 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; and (2) Java can achieve almost similar performance to native languages, both for sequential and parallel applications, being an alternative for HPC programming. 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.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; C.4 [**Performance of Systems**]: Performance attributes, Measurement techniques; C.2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks—*High-speed, Ethernet*

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]. 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, 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 adoption of Java in HPC is being delayed by the lack of analysis of the existing programming options in this area and 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 embracement.

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 [12]. 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 and future work.

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 Cyclic-Barrier.

The project Parallel Java (PJ) [17] 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 [16] and JaMP [18]. 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 [25]. 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 equiva-

lent 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 [33], 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 [34] 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 [11], 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 [29], 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 [22], a parallel and distributed Java computing framework. However, their implementation on top of the JVM sockets library limits their performance benefits.

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 [2] 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 [23], RMIX [19], Manta [20], Ibis RMI [22], and Opt RMI [27]. 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 [28]). 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 [8], 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 [9], which is the proposal of the Java Grande Forum (JGF) [15] 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 [3] 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 [25], 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).

Currently, these two projects, mpiJava and MPJ Express, are the most active projects in terms of uptake by the HPC community, presence on academia and production environments, and available documentation. These projects are also stable and publicly available along with their source code.

In order to update the compilation of Java message-passing implementations presented in [28], this paper presents the projects developed since 2003, in chronological order:

- MPJava [24] is the first Java message-passing library implemented on Java NIO sockets, taking advantage of their scalability and high performance communications.

- Jcluster [35] 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) [17] 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 difficults its adoption.

- P2P-MPI [13] 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 [6] 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 [4] 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) [30] is our scalable Java message-passing implementation which provides high-speed networks support (see Section 3).

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		
	Pure Java Impl.	Java IO	Java NIO	Myrinet	InfiniBand	SCI	mpiJava 1.2	JGF MPJ	Other APIs
MPJava [24]	✓		✓						✓
Jcluster [35]	✓	✓							✓
Parallel Java [17]	✓	✓							✓
mpiJava [3]				✓	✓	✓	✓		
P2P-MPI [13]	✓	✓	✓				✓		
MPJ Express [25]	✓		✓	✓			✓		
MPJ/Ibis [6]	✓	✓		✓				✓	
JMPI [4]	✓	✓							✓
F-MPJ [30]	✓	✓		✓	✓	✓	✓		

3. JAVA FOR HPC: CURRENT RESEARCH

This section describes current research efforts in Java for HPC, which can be classified into: (1) development of high performance Java sockets for HPC; (2) design and implementation of low-level Java message-passing devices; (3) improvement of the scalability of Java message-passing collective primitives; and (4) implementation and evaluation of efficient MPJ benchmarks. 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.

3.1 High Performance Java Sockets

Java Fast Sockets (JFS) [29] is our high performance Java socket implementation for HPC, available at http://jfs.des.udc.es. 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 (see Figure 1); (2) supporting high performance native sockets communication over SCI Sockets, Sockets-MX, and Socket Direct Protocol (SDP), on SCI, Myrinet and InfiniBand, respectively (see Figure 2); (3) avoiding the need of primitive data type array serialization; and (4) reducing buffering and unnecessary copies. Its interoperability and user and application transparency through reflection allow for immediate applicability on a wide range of parallel and distributed target applications.
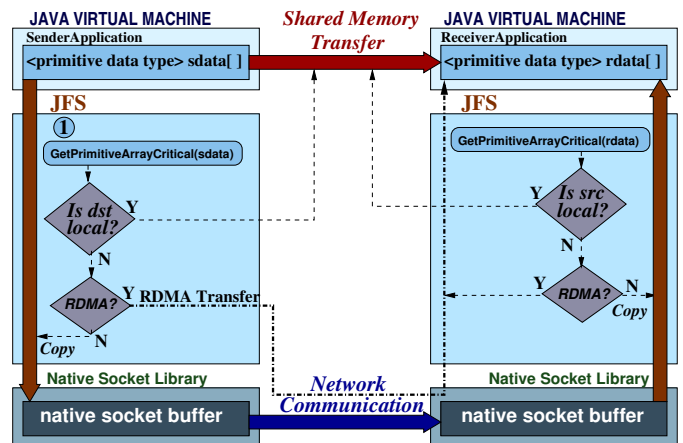

Figure 1: JFS optimized protocol

The avoidance of primitive data type serialization is provided by JFS extending the sockets API in order to allow the direct sending of primitive data type arrays (e.g., jfs.net.SocketOutputStream.write(int buf[], int offset, int length)). In the implementation of these read-/write socket stream methods it has been used the JNI function GetPrimitiveArrayCritical(<primitive data type> sdata[]) (see point (1) in Figure 1), which allows native code to obtain, through JNI, a direct pointer to the Java array, thus avoiding serialization. Therefore, a one-copy protocol can be implemented in JFS, as only one copy is needed to transfer sdata to the native socket library.

JFS reduces significantly JVM sockets communication overhead (see Table 2). According to Figure 1, JFS needs up to two data copies and a network communication, or only a shared memory transfer. JVM IO sockets can involve up to nine steps (see [29]): a serialization, three copies in the sender side, a network transfer, another three copies in the receiver side, and a deserialization.

Table 2: JFS performance improvement compared to Sun JVM sockets

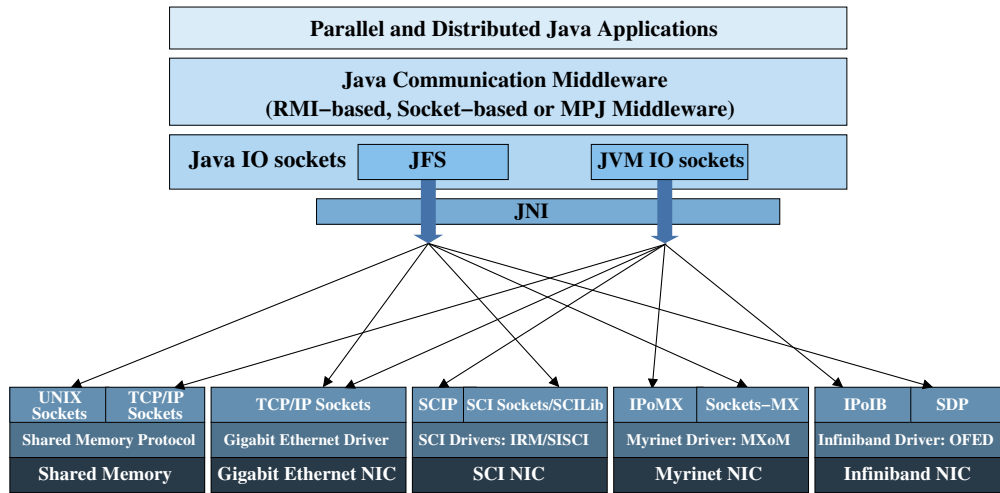|	JFS start-up reduction	JFS bandwidth increase
Shared memory	up to 50%	up to 4411%
Gigabit Ethernet	up to 10%	up to 119%
SCI	up to 88%	up to 1305%
Myrinet	up to 78%	up to 412%
InfiniBand	up to 65%	up to 860%

Figure 2: Java communication middleware on high-speed multi-core clusters

JFS transparency is achieved through Java reflection: the built-in procedure (setting factories) to swap the default socket library can be used in a small application, *launcher*, which invokes, through Java reflection, the `main` method of the target Java class (see Listing 1). This target Java application will use JFS transparently from then on, even without source code availability. Finally, JFS is portable because it implements a general pure Java solution over which JFS communications can rely on absence of native communication libraries, although it obtains, in general, worse performance than the native approach.

Listing 1: JFS *launcher* application code

```
SocketImplFactory factory;
factory = new jfs.net.JFSImplFactory();
Socket.setSocketImplFactory(factory);
ServerSocket.setSocketFactory(factory);

Class cl = Class.forName(className);
Method method = cl.getMethod("main",parametrTypes);
method.invoke(null, parameters);
```

3.2   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 Open-MPI include several devices on Myrinet, InfiniBand and shared memory. Regarding MPJ libraries, in MPJ Express the low-level xdev layer [25] provides communication devices for different interconnection technologies. The two implementations of the xdev API currently available are niodev (over Java NIO sockets) and mxdev (over Myrinet MX). Furthermore, there are two shared memory xdev implementations [26], one thread-based (pure Java) and the other based on native IPC resources, and two more xdev devices are being implemented, one on native MPI implementations and the other on InfiniBand. This latter can take full advantage of the low-level InfiniBand Verbs layer, like Jdib [14].

Additionally, we have implemented a low-level communication device based on Java IO sockets which presents an API similar to xdev [30]. The motivation behind this development is the research on the efficiency of Java message-passing protocols based on Java IO sockets. Thus, this device, iodev, can run on top of JFS, and hence obtain high performance on shared memory and Gigabit Ethernet, SCI, Myrinet, and InfiniBand networks. In order to evaluate the impact of iodev on MPJ applications we have implemented our own MPJ library, Fast MPJ (F-MPJ) [30], on top of iodev.

3.3   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 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 [5, 10, 31, 32], but not in MPJ. Therefore, in F-MPJ we have adapted the research in native libraries to Java. As far as we know, this is the first project in this sense, as up to now MPJ library developments have been focused on providing production-quality implementations of the full MPJ specification, rather than concentrate on developing scalable MPJ collective primitives.

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 (Four-aryT), Bucket (BKT) or cyclic, and BiDirectional Exchange (BDE) or recursive doubling, which are extensively described in [10]. Table 3 presents a complete list

of the collective algorithms used in F-MPJ and MPJ Express (the prefix "b" means that only blocking point-to-point communication are used, whereas "nb" that non-blocking primitives are used). It can be seen that F-MPJ implements up to three algorithms per collective primitive, allowing their selection at runtime, as well as it takes more advantage of communications overlapping, achieving higher performance scalability. As MPJ libraries (e.g., MPJ Express) can benefit significantly from the use of these collective algorithms we plan to distribute soon our MPJ collectives library implementation.

Table 3: Collective algorithms used in representative MPJ libraries ([1]selected algorithm for short messages; [2]selected algorithm for long messages; [3]selectable algorithm for long messages and number of processes power of two)

Collective	F-MPJ	MPJ Express
Barrier	MST	nbFTGather+ bFour-aryTBcast
Bcast	MST[1] MSTScatter+BKTAllgather[2]	bFour-aryT
Scatter	MST[1] nbFT[2]	nbFT
Scatterv	MST[1] nbFT[2]	nbFT
Gather	MST[1] nbFT[2]	nbFT
Gatherv	MST[1] nbFT[2]	nbFT
Allgather	MSTGather+MSTBcast[1] BKT[2] / BDE[3]	nbFT
Allgatherv	MSTGatherv+MSTBcast	nbFT
Alltoall	nbFT	nbFT
Alltoallv	nbFT	nbFT
Reduce	MST[1] BKTReduce_scatter+ MSTGather[2]	bFT
Allreduce	MSTReduce+MSTBcast[1] BKTReduce_scatter+ BKTAllgather[2] / BDE[3]	BT
Reduce_- scatter	MSTReduce+MSTScatterv[1] BKT[2] / BDE[3]	bFTReduce+ nbFTScatterv
Scan	nbFT	nbFT

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 [7], 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) [21], 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 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	Medium		✓

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%). 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.

4. PERFORMANCE EVALUATION

This section presents an up-to-date comparative evaluation of current Java and native solutions for HPC using the NPB on two representative scenarios: a multi-core InfiniBand cluster and a 24-core shared memory machine.

4.1 Experimental Configuration

The InfiniBand cluster consists of eight dual-processor nodes (Pentium IV Xeon 5060 dual-core, 4 GB of RAM) interconnected via InfiniBand dual 4X NICs (16 Gbps). The InfiniBand driver is OFED 1.4, and the MPI implementation is Intel 3.2.0.011 with InfiniBand support. The performance

results on this system have been obtained using one core per node, except for 16 and 32 processes, for which two and four cores per node, respectively, have been used. The shared memory machine has four Pentium IV Xeon 7450 hexa-core processors (hence 24 cores) and 32 GB of RAM. The benchmarks have been evaluated using up to the number of available cores on this system (24). Both scenarios share the remaining configuration details. The OS is Linux CentOS 5.1, the C/Fortran compiler (with -fast flag) is the Intel version 11.0.074 with OpenMP support, and the JVM is Sun JDK 1.6.0_05. The evaluated MPJ libraries are F-MPJ with JFS 0.3.1, MPJ Express 0.27 and mpiJava 1.2.5x. It has been used the NPB-MPI/NPB-OMP version 3.3 and the NPB-JAV version 3.0. The ProActive version used is the 4.0.2, which includes its own implementation of the NPB (NPB-PA). The metric that has been considered is MOPS (Millions of Operations Per Second), which measures the operations performed in the benchmark, that differ from the CPU operations issued. Moreover, Class B workload has been used as their performance is highly influenced by the efficiency in communications, both the network interconnect and the communication library. Therefore, the differences among parallel libraries can be appreciated more easily.

4.2 Experimental Results on One Core

Figure 3 shows a performance comparison of several NPB implementations on one Xeon 5060 core. 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. Only F-MPJ results are shown for NPB-MPJ performance for clarity purposes, as other MPJ libraries obtain quite similar results on one core. Here, the differences that can be noted are explained by the different implementations of the NPB benchmarks, and the use of Java or native code (C/Fortran). Thus, the ProActive code generally obtains good results due to its efficient implementation, especially for EP and IS, whereas Java FT implementations achieve poor performance. Java Threads EP and ProActive SP results are missing from Figure 3 as these kernels are not implemented in their respective NPB suites (NPB-JAV and NPB-PA).
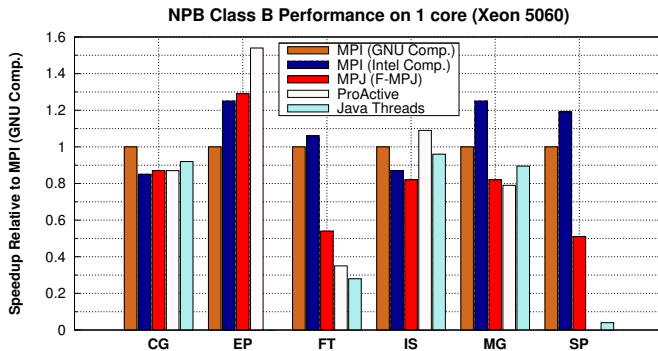

Figure 3: NPB relative performance on one core

NPB-MPI results have also been obtained with the GNU compiler version 4.1.2. As the publicly available Sun JVM for Linux has been built with the GNU compiler, Java performance is limited by this compiler throughput. Thus, the Java results of Figure 3 (MPJ, ProActive and Java threads) are usually slightly lower than those of GNU-built benchmarks, although it is possible that Java benchmarks outperform native code (EP), or, on the contrary, obtain around half of the native performance (FT and SP). From now on only the Intel compiler results are shown as it usually outperforms GNU compiler.

4.3 Java Performance for HPC

Figure 4 shows NPB-MPI, NPB-MPJ and NPB-PA performance on the InfiniBand cluster, and NPB-OMP and NPB-JAV on the 24-core shared memory machine. Although the configuration of the shared and the distributed scenarios are different, their results are shown together in order to ease their comparison. Moreover, the NPB-MPJ results have been obtained using three MPJ libraries: mpiJava, MPJ Express and F-MPJ, in order to compare them.

Regarding CG results, NPB-PA, NPB-OMP and NPB-JAV, due to their inefficient benchmark implementations, show the lowest performance, whereas MPJ libraries achieve high performance, especially mpiJava and F-MPJ. In fact, mpiJava outperforms MPI up to 16 cores. EP presents low communication intensiveness (see Table 4). Thus, speedups almost linear are expected. In this case NPB-OMP achieves the highest performance, followed by NPB-PA. The remaining libraries obtain quite similar performance among them. Within FT results the native solutions show the highest performance, NPB-OMP up to 8 cores and NPB-MPI on 16 and 32 cores. Among Java results F-MPJ achieves the highest performance, around 15% lower than MPI on 32 cores, whereas NPB-PA shows the lowest results up 16 cores. Moreover, shared memory solutions do not take advantage of the use of more than 8 cores.

The communication intensiveness of IS reduces Java performance, except for NPB-JAV up to four cores. Regarding native implementations, OpenMP obtains the best results up to 8 cores, whereas MPI achieves the highest scalability and performance from 16 cores. The highest MG performance has been obtained with NPB-MPI and NPB-MPJ with mpiJava, whereas the lowest with NPB-PA and shared memory programming.

The NPB-MPJ SP benchmark obtains generally high performance, especially for F-MPJ and mpiJava, outperforming significantly shared memory programming, especially NPB-JAV. In this case MPJ scalability is higher than that of MPI as MPJ on one core achieved around half of the performance of MPI, but on 32 cores rises up to 75% of MPI results. In this case, mpiJava and F-MPJ shows similar performance among them. A particular feature of SP is that it requires a square number of processes (1, 4, 9, 16, 25...). On the InfiniBand cluster it is used one core per node up to 9 processes, two cores per node for 16 processes and three cores per node for 25 processes. In this scenario all distributed memory options (MPI and MPJ) take advantage of the use of up to 16 cores, whereas NPB-OMP obtains the highest performance on 9 cores.
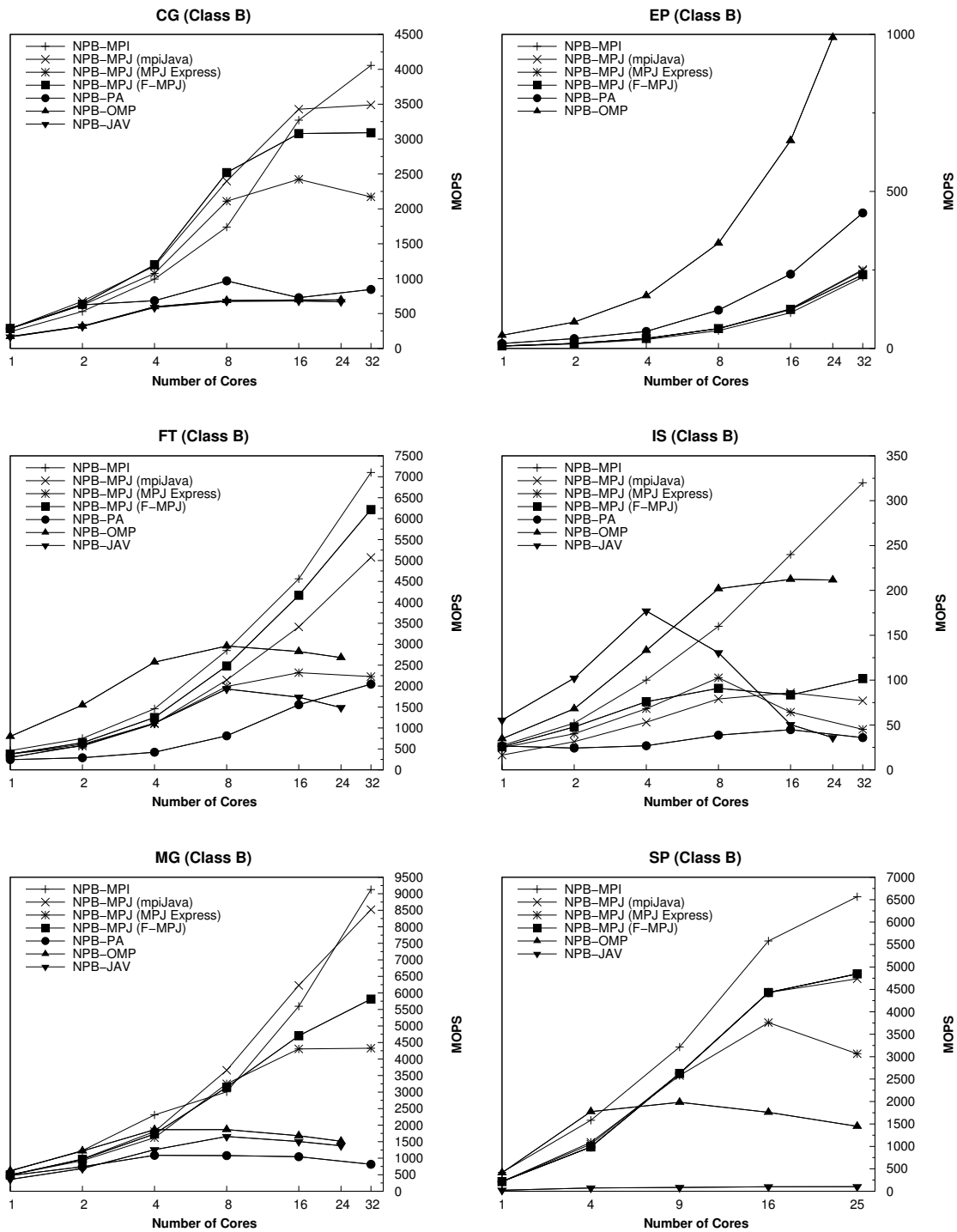
Figure 4: NPB Class B results

These NPB experimental results can be analyzed in terms of the three main evaluations that NPB-MPJ allows. The first one is the comparison among MPJ implementations, which present quite significant differences in performance, except for EP, due to their communication efficiency. Thus, in this testbed, MPJ Express uses IPoIB, obtaining relatively low performance, F-MPJ relies on the InfiniBand support of JFS, implemented on SDP, thus achieving much higher speedups. Finally, mpiJava relies on the high performance MPI support on InfiniBand, in this case implemented on IBV (InfiniBand Verbs). CG and MG results confirms the highest performance of mpiJava compared to MPJ Express and F-MPJ. However, F-MPJ obtains the best MPJ performance for IS, FT and SP, showing that it is possible to achieve significant performance benefits without the drawbacks of mpiJava. Finally, MPJ Express achieves good results on CG and SP, thanks to the efficient non-blocking support provided by Java NIO.

The second evaluation that can be performed is the comparison of MPJ against other Java parallel libraries, in this case ProActive and Java threads. ProActive is an RMI-based middleware, and for this reason its performance is usually lower than that of MPJ libraries, whose communications are based on MPI or on Java sockets. Moreover, ProActive does not support InfiniBand in our testbed (neither on SDP nor IPoIB), so it resorted to Gigabit Ethernet. Thus, its scalability was significantly worse than that of NPB-MPJ results. Regarding Java threads, NPB-JAV only obtains good results for IS up to 4 cores.

Finally, NPB-MPJ allows the comparative performance evaluation of MPJ against MPI. Except for CG and IS, the gap between Java and native performance narrows as the number of cores grows. This higher MPJ scalability helps to bridge the gap between Java and native code performance.

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. The performance evaluation of existing Java solutions and research developments in Java for HPC on a multi-core InfiniBand cluster, and on a 24-core shared memory machine allows us to conclude that Java can achieve almost similar performance to native 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. Finally, the active research efforts in this area are expected to bring in the next future new developments that will bridge the gap with native performance and will increase the benefits of the adoption of Java for HPC.

Acknowledgments

6. REFERENCES

[1] B. Amedro, V. Bodnartchouk, D. Caromel, C. Delbé, F. Huet, and G. L. Taboada. Current State of Java for HPC. In *INRIA Technical Report RT-0353*, 24 pages, http://hal.inria.fr/inria-00312039/en/ [Last visited: July 2009].

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

[3] M. Baker, B. Carpenter, G. Fox, S. Ko, and 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*, pages 748–762, San Juan, Puerto Rico, 1999.

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

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

[6] M. Bornemann, R. V. v. Nieuwpoort, and 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*, pages 217–224, Sorrento, Italy, 2005.

[7] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A Benchmark Suite for High Performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, 2000.

[8] B. Carpenter, G. Fox, S.-H. Ko, and S. Lim. mpiJava 1.2: API Specification. http://www.hpjava.org/reports/mpiJava-spec/mpiJava-spec/mpiJava-spec.html [Last visited: July 2009].

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

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

[11] K. Datta, D. Bonachea, and 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*, pages 200–214, Hawthorne, NY, USA, 2005.

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

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

[14] W. Huang, H. Zhang, J. He, J. Han, and L. Zhang. Jdib: Java Applications Interface to Unshackle the

Communication Capabilities of InfiniBand Networks. In *Proc. 4th Intl. Conf. Network and Parallel Computing (NPC'07)*, pages 596–601, Dalian, China, 2007.

[15] Java Grande Forum. http://www.javagrande.org. [Last visited: July 2009].

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

[17] 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)*, page 196a (8 pages), Long Beach, CA, USA, 2007.

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

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

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

[21] D. A. Mallón, G. L. Taboada, J. Touriño, and 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)*, pages 181–190, Weimar, Germany, 2009.

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

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

[24] B. Pugh and 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*, pages 323–339, College Station, TX, USA, 2003.

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

[26] A. Shafi and J. Manzoor. Towards Efficient Shared Memory Communications in MPJ Express. In *Proc. 11th Intl. Workshop on Java and Components for Parallelism, Distribution and Concurrency (IWJacPDC'09), Rome, Italy*, page 111b (8 pages), 2009.

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

[28] G. L. Taboada, J. Touriño, and 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)*, pages 118–126, Hong Kong, China, 2003.

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

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

[31] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *Intl. Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[32] S. S. Vadhiyar, G. E. Fagg, and J. J. Dongarra. Towards an Accurate Model for Collective Communications. *Intl. Journal of High Performance Computing Applications*, 18(1):159–167, 2004.

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

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

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