ProActive: Using a Java Middleware for HPC Design, Implementation and Benchmarks

Brian Amedro, Denis Caromel, Fabrice Huet
INRIA Sophia-Antipolis, CNRS, I3S, UNSA.
2004, Route des Lucioles, BP 93
06902 Sophia-Antipolis Cedex, France.
First.Last@inria.fr

Vladimir Bodnartchouk, Christian Delbé
ActiveEon
2004, Route des Lucioles, BP 93
06902 Sophia-Antipolis Cedex, France
First.Last@activeeon.com

Guillermo L. Taboada
University of A Coruña
Faculty of Informatics, Spain
taboada@udc.es

***Abstract*—Although Java is among the most used programming languages, its use for HPC applications is still marginal. This article reports on the design, implementation and benchmarking of a Java version of the NAS Parallel Benchmarks translated from their original Fortran / MPI implementation. We have based our version on ProActive, an open source middleware designed for parallel and distributed computing. This paper gives a description of the ProActive middleware principles, and how we have implemented the NAS Parallel Benchmark on such Java library. We Also gives some basic rules to write HPC code in Java. Finally, we have compared the overall performance between the legacy and the Java ProActive version. We show that the performance varies with the type of computation but also with the Java Virtual Machine, no single one providing the best performance in all experiments. We also show that the performance of the Java version is close to the Fortran one on computational intensive benchmarks. However, on some communications intensive benchmarks, the Java version exhibits scalability issues, even when using a high performance socket implementation (JFS).**

I. Introduction

Message Passing Interface (MPI) is the dominant programming model of choice for scientific computing. This library proposes many low-level primitives designed for pure performance. But for several years, the tendency has been to look for productivity[13], and to propose efficient high-level primitives like collective operations [9], object-oriented distributed computing [6] and material to ease the deployment of applications.

In order to perform an evaluation of Java capabilities for high performance computing, we have implemented the *NAS*[1] *Parallel Benchmarks* (NPB) which are a standard in distributed scientific computation. Many middleware comparatives and optimization techniques are usually based on them [17], [7], [10], [12], [8]. They have the characteristic to test a large set of aspects of a system, from pure computation performance to communication speed.

By using a Java-based middleware, instead of Fortran+MPI, we want to demonstrate the performance which can be obtained, comparing it to an equivalent native version. Our aim is to identify the areas where Java still lacks some performance, in particular the network layer.

[1]Numerical Aerodynamic Simulation

Our contributions are the following :

- An evaluation of the Java overhead for arithmetic computation and array manipulation
- A report on common performance pittfals and how to avoid them
- A performance comparison of an implementation of the NPBs in Java and Fortran/MPI (*PGI*) on Gigabit Ethernet and SCI

The rest of this paper is organized as follows. Section 2 gives some background: a short description about the benchmarks used in our experiments, the ProActive library (in particular the active object model), and the *Java Fast Sockets*[18]. Section 3 presents some related work. In section 4, we discuss the implementation and some performance issues. Section 5 presents the results obtained with the NAS Parallel Benchmarks on two network architectures. Finally, we discuss the future work and conclude in section 5.

II. Background

A. The NAS Parallel Benchmarks

NAS Parallel Benchmarks (NPB) consists of a set of kernels which are derived from computational fluid dynamics (CFD) applications. They were designed by the NASA Ames Research Center and test different aspects of a system.

Some are testing pure computation performance with different kinds of problems like matrix computation or FFTs. Others involve a high memory usage or network speed with large data size communications. Finally, some problems try to evaluate the impact of irregular latencies between processors (short or long distance communications). Each of these five kernels was designed to test a particular subset of these aspects. To follow the evolution of computer performance, the NPB were designed with several classes of problems making kernels harder to compute by modifying the size of data and/or the number of iterations. There are now 6 classes of problems: S, W, A, B, C and D. Class S is the easiest problem and is for testing purpose only. Class D is the hardest and usually requires a lot of memory.

Here we will use the IS, FT, EP, CG and MG kernels with the problem class C.

B. The ProActive Library

ProActive is a GRID middleware (a Java library with open source code under LGPL license) for parallel, distributed, and concurrent computing in a uniform framework. With a reduced set of simple primitives, ProActive provides a comprehensive API to simplify the programming of Grid Computing applications: distributed on Local Area Network (LAN), on clusters of workstations, or on Internet Grids.

ProActive uses standard RMI as a transport layer and is thus bound to its limitations [11]. However, the RMI transport overhead can be reduced through the use of a high performance Java sockets implementation named Java Fast Sockets (*JFS*)[18]. *JFS* provides high performance network support for Java (currently direct Scalable Coherent Interface –SCI– support). It also increases communication performance avoiding unnecessary copies and buffering, by reducing the cost of primitive data type array serialization, the process of transforming the arrays in streams to send across the network.

Although our implementation of the NPBs uses some ProActive specific features, it could easily be ported to another middleware. Thus the insights gain from these experiments will be valuable to the HPC community, irrespective of their use of ProActive.

ProActive is only made of standard Java classes, and requires no changes to the Java Virtual Machine, no preprocessing or compiler modification; programmers write standard Java code. Based on a simple Meta-Object Protocol, the library is itself extensible, making the system open for adaptations and optimizations. ProActive currently uses the RMI[2] Java standard library as default portable transport layer.

The *active object* is the base entity of a ProActive application. It is a standard Java object which has its own thread. An active object can be created on any host used for the application deployment. Its activity and localization (local or distant) are completely transparent. Asynchronous requests are sent to an active object. These requests are stored in the active object's *request queue* before being served according to a *service policy*. By default, this service policy is FIFO, but the user can create its own. A *future object* is a place holder for the asynchronous request result with a *wait-by-necessity* mechanism for synchronization. Some requests can be invoked as *immediate services*: these requests will be served in parallel of the main thread and other *immediate services*. For further details about ProActive, the reader can refer to [4].

C. High Performance Java Sockets Support in ProActive

As the use of Java RMI as transport layer in Proactive has an important impact on performance, it has been considered the substitution of the current RMI transport protocol by a more efficient one. The RMI transport overhead can be reduced through the use of a high performance Java sockets implementation named Java Fast Sockets (*JFS*)[18]. *JFS* provides high performance network support for Java (currently direct Scalable Coherent Interface –SCI– support). It also increases communication performance avoiding unnecessary copies and buffering, by reducing the cost of primitive data type array serialization, the process of transforming the arrays in streams to send across the network. Most of these optimizations are based on the use of native methods as they obtain higher performance. On SCI, *JFS* makes JNI calls to SCI Sockets, SCILib and SISCI, three native communication libraries on SCI. *JFS* increases communication throughput looking for the most efficient underlying communication library in every situation. Moreover, it is portable because it implements a general "pure" Java solution over which *JFS* communications can rely on absence of native communication libraries. The "pure" Java approach usually leads in the lowest performance, but the stability and security of the application (associated trade-offs for the higher performance of the native approach) is not compromised. The transparency to the user is achieved through Java reflection: the *Factory* for creating Sockets can be set at application start-up to the *JFS SocketImplFactory*, and from then on, all sockets communications will use *JFS*. This process can be done in a small Java application launcher that will call the target Java application. This feature allows any Java application, and in this particular case ProActive middleware, to use *JFS* transparently and without any source code modification.

[2]Remote Method Invocation

III. Related Work

Studies of Java for High Performance Computing can be traced back to the JavaGrande Forum community effort[16]. The results, at that time, were disappointing and gave Java a bad reputation. Since then, only a few works have been dedicated to this task, although the technologies behind the Java Virtual Machines and the computer architecture have changed a lot over the years. A notorious performance hit was the garbage collector, because of the pauses it introduced. Nowadays, all JVMs come with multiple garbage collectors implementation which can be chosen at start-time [2], [1]. Multi-core CPUs are now mainstream and might change fundamentally the performance of Java. Indeed, a JVM is multi-threaded and can take advantage of multiple cores to perform background tasks like memory management or Just-In-Time compilation.

A recent work is the DaCapo Benchmarks suite [5]. The authors define a set of benchmarks and methodologies meaningful for evaluating the performance of Java. As noted by the authors, Java introduces complex interactions between the architecture, the compiler, the virtual machine and the memory management through garbage collectors. As such, using benchmarks and methodologies developed for Fortran/C/C++ might put Java at a disadvantage.

IV. Implementation

Our implementation of the NPB is done strictly in standard Java, without relying on external libraries except for communication. As we will show in this section, writing HPC code in Java is possible but requires care and good knowledge of the internals of the JVMs. Also, we have tried to be as

close as possible to the original NPB3.2-MPI implementation. However, there are a few dissimilarities induced by both the object oriented model (Java) and the distribution library (ProActive).

Using the Object-Oriented SPMD layer provided by the ProActive library [3], each SPMD MPI process has been translated to an *active object* (remotely accessible Java object) named `Worker`. Due to the ProActive principles, we have also redefined the iterations in tail-recursive calls. Thus, each kernel iteration is a *ProActive request*.

Table I gives a brief overview of the equivalences between MPI and ProActive operations.

MPI	**ProActive**
`mpirun`	*deployment*
`MPI_Init` `MPI_Finalize`	*activities creation*
`MPI_Comm_Size`	`getMyGroupSize`
`MPI_Comm_Rank`	`getMyRank`
`MPI_*Send` `MPI_*Receive`	*method call* *(setter and getter)*
`MPI_*Sendrecv`	`exchange`
`MPI_Barrier`	`barrier`
`MPI_Bcast`	*method call on a group communication*
`MPI_Scatter`	*method call with a scatter group*
`MPI_Gather`	*result of a group communication*
`MPI_Reduce`	*programmer's method*

TABLE I
TRANSLATION BETWEN MPI AND PROACTIVE

A. Exchange Operator

Due to its request queue, ProActive is not able to translate directly some constructions which are frequently used with MPI, like the *irecv() / send() / wait()* sequence. With MPI, this sequence offers a way to exchange potentially a very large amount of data between 2 processes, making an implicit synchronization. With ProActive, the asynchronous request service mechanism is not really adapted to such task and should lead to some dead-locks, involving, in addition to an explicit synchronization, to make a copy of data to exchange (received data will first be copied into a request which will be served later).

In order to avoid unnecessary copying of data, we take advantage of the serialization step of a request, through a special *exchange* operator. It allows two ProActive processes to exchange data efficiently. The two processes will invoke the *exchange* operator in order to exchange data with each other. It is equivalent to a pair of `irecv() / send() / wait()` calls by those two processes. Hence, the ProActive call:

```
exchange(tag,destRank,srcArray,srcOffset,
         destArray,destOffset, len)
```

is equivalent to the MPI sequence:

```
mpi_irecv(dest_array,len,data_type,
          dest_rank, tag,mpi_comm_world,
          req)
mpi_send(src_array,len,data_type,
         dest_rank,tag,mpi_comm_world)
mpi_wait(req,status,ierr)
```

Figure 1 explains the inner working of the operator: In this example, two active objects (AOs) are exchanging a subset of an array using the *exchange* operator, which works like a 2-way array copy. Thus, both AOs creates a $RequestExchange$ object containing the details of the source and destination arrays (ie. array pointers, offsets and length). Then, each AO sends its $RequestExchange$ object to the other one, performing an implicit synchronization when serializing and deserializing data. Indeed, the serialized data are directly read in the source array without any intermediate copy. In the same way, deserialized data are directly put in the destination array without any intermediate copy.

Moreover the *Exchange* does not work exclusively with arrays of primitives, but also with any complex data structure which implements the $java.util.Collection$ interface.

B. Basic arithmetic operations

Some of the primitives provided by the standard JVM for numerical operations are not very efficient. Especially, simple operations such as integer binary logarithm or binary powering are not optimized by either the static compiler or the JIT.

For example, on the Table IV-B, we compare performance between standard and optimized functions for pow and log. It shows that HPC programmer should take care about arithmetic operations and consider optimizing by-hand some of its intensive computation loops. For efficient and higher level mathematical computation, developer can use specialized libraries such as MKL[3] which provides efficient primitives for operations such as matrix computation and fast fourier transformations.
However, due to license limitations, we did not use MKL in our implementation, but only rewrote basic operations with base-2 and integer optimizations. Note that is has been done only in innermost loops.

C. Optimization of data structures memory footprint

HPC applications are often characterized by the use of large data structures. Thus, developer of such application might be aware of the actual memory footprint of the objects he deal with.

Java offers two ways to store data: primitive types, and Objects. Primitive types only contain *actual data* whereas

[3]Math Kernel Library

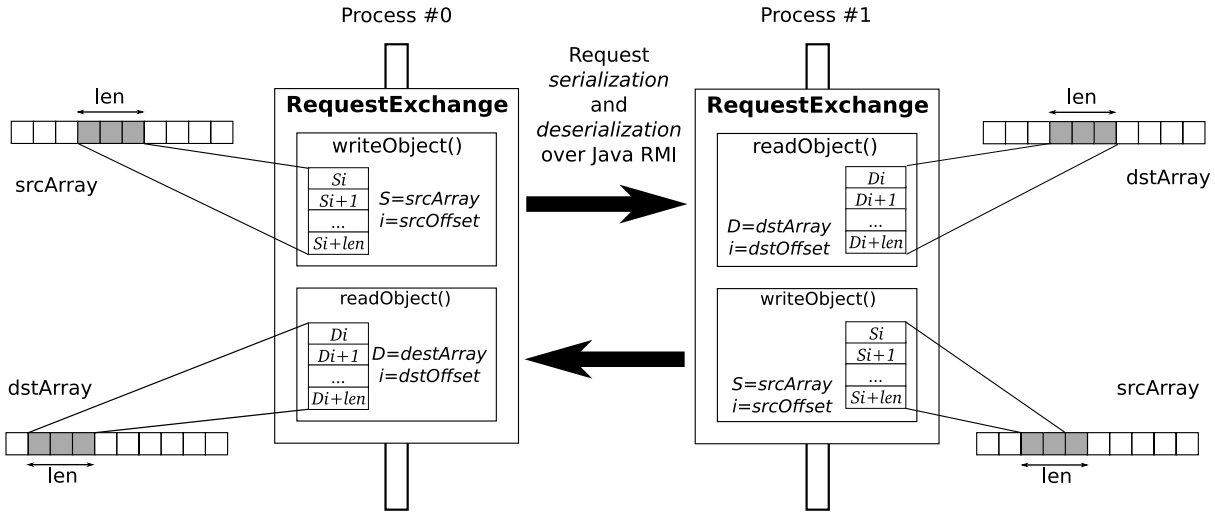

Fig. 1. Data exchange operation between 2 Active Objects in ProActive

	Sun 1.6	IBM 1.6	BEA 1.6	gcc 4.3.2
pow(2,i)	305.28	256.58	201.82	107.5
$1 << i$	8.32	9.06	10.99	8.5
log(i)/log2	90.42	168.51	91.35	92.3
ilog2(i)	11.18	13.12	16.89	19.3

TABLE II
PERFORMANCE COMPARISON BETWEEN STANDARD ARITHMETIC FUNCTIONS AND OPTIMIZED VERSIONS (VALUES ARE IN SECONDS)

	Sun 1.6	IBM 1.6	BEA 1.6
byte[]	10 MB	10 MB	10 MB
Byte[]	80 MB	80 MB	40 MB
short[]	20 MB	20 MB	20 MB
Short[]	320 MB	320 MB	200 MB
int[]	40 MB	40 MB	40 MB
Integer[]	320 MB	320 MB	200 MB
double[]	80 MB	80 MB	80 MB
Double[]	320 MB	320 MB	200 MB

TABLE III
MEMORY FOOTPRINT COMPARISON BETWEEN PRIMITIVE TYPES AND OBJECT TYPE WRAPPERS ON A 10 MILLIONS ELEMENT ARRAY WITH DIFFERENT JVM VENDORS ON 64 BITS ARCHITECTURE

Objects have associated meta-data which are used by the JVM to enforce language properties or features. For every primitive type, there is an equivalent Object which acts as a wrapper (double and Double, int and Integer...).

As shown on Table IV-C, there is an important difference between primitive types and Objects. For this comparison we have used a large array of 10 millions elements and measured the memory usage. Although the Oracle JRockit 1.6 JVM needs less memory for the same amount of data, Object payload is important compared to primitive, especially for integer or double. Thus, handling large data structures requires using primitive types.

Allocation strategy of multi-dimensional arrays also has an important impact on memory footprint, because Java does not have support for true multidimensional arrays. Instead, it relies on arrays of arrays to simulate them. This leads to non rectangular arrays with variable shapes. Also, Java arrays are actually Objects even if they only contain primitive type data. It is very hard for a compiler to perform optimizations on such array. Although some solutions have been proposed to address these issues [15], none has made it in the official releases of Java.

Table IV-C shows the memory usage of various JVMs when allocating a 2-dimensions array of 20M elements (double or byte). We also indicate the value measured on a C and a Fortran versions compiled with gcc. We have measured 3 different allocations strategies : `[2][10M]`, `[10M][2M]` and `[20M]`. As expected, the lowest usage is obtained when allocating a single dimension array, as the memory can be allocated contiguously in memory. When allocating two-dimensional arrays, we see that the Oracle JRockit JVM performs sometimes better than the C version. We believe this is because when instantiating the array in the Java version, the bounds are known and thus the JVM has enough information to manage memory in a more efficient way. For its part, Fortran inlines all multi-arrays, regardless of the allocation strategy, thus allowing optimal memory management.

	Sun 1.6	IBM 1.6	BEA 1.6	gcc 4.3.2	f77
byte[2][10M]	19.1	19.1	19.1	19.2	19.1
byte[10M][2]	380	381	269	381	19.1
byte[2*10M]	19.1	19.1	19.1	19.2	19.1
dble[2][10M]	152	152	152	152	152
dble[10M][2]	456	457	345	381	152
dble[2*10M]	152	152	152	152	152

TABLE IV
MEMORY FOOTPRINT COMPARISON ON MULTI-ARRAY DECLARATION STRATEGIES (IN MB) ON 64 BITS ARCHITECTURE

As we can see, to be memory-efficient we have to use one-dimensional arrays. Also, In the Fortran implementation of the NPBs, one dimensional arrays are often seen as 2 or 3 dimensional one. However, there is no direct support in Java for such operations. When necessary, we have rewritten the Java code to manipulate only one dimensional arrays, using a simple flattening technique and adding methods to treat them as multidimensional one.

The first versions of Java suffered from automatic bounds checking of arrays. However, since the NPBs operate on arrays with known size at runtime, most of the unnecessary checks are removed by the Just-In-Time compiler [20].

D. JIT

Compared to Fortran or C, most of the optimization in Java are not performed at compile time but at run time, by the Just-In-Time compiler (JIT) [14] which usually comes in two flavors: client and server. The main difference being that the second one performs more aggressive optimization and might incur a higher overhead. The decision of compiling a method is mainly based on the number of invocations already performed or the number of backward branches taken in loops (both controlled by the *CompileThreshold* property). One of the difficulties is to write code which will lead to high performance after being compiled by the JIT. Thus, as the JIT compiler mostly works on methods, our experience in the development of the NPBs have confirmed that keeping small methods (i.e avoiding inlining) lead to better performance.

V. Experimentation

A. Experimentation Methodology

We divide the five kernels in two categories. If the kernel performs many calls with a particular communication scheme, we define it as a *communication intensive* one; otherwise, it is a *computation intensive* one. Following this study, each kernel was run with different parameters:

- the JVM version and vendor: BEA (5 and 6), IBM (5 and 6) and Sun (5, 6 and 7),
- the initial and maximum heap size of the JVM,
- the number of nodes used (from 1 to 32),
- the kernel problem class size (class S or C)
- the network architecture (GbE or SCI)

Some values or combinations had no impact on the running of the NPBs and are not presented in the remaining of the paper. Also, to minimise the mean error, all the presented values are the average of at least five runs.

The NPB ProActive implementation we have developed is based on the NPB 3.2 version distributed by NASA. The Fortran MPI version was compiled with the 64 bits PGI 7.1 compiler and run onto a MPICH 2. For our experiments, we have used two clusters, described on Tab V.

On the SCI cluster, experiments have been run using one process per node (single process configuration) or four processes per node (quad process configuration). The transport protocol for ProActive on the SCI cluster is JFS, which achieves a latency of 6 microseconds and an asymptotic throughput of 2398 Mbps. The native MPI library presents a latency of 4 microseconds and an asymptotic throughput of 2613 Mbps. Thus, this high-performance interconnect cluster can achieve significantly higher performance scalability.

	GbE Cluster
Processor	AMD Opteron 2218 2.6 GHz / 2x1 MB L2 cache / 667 MHz 50 nodes x 1 cpu per node = 50 cpus 50 cpus x 4 cores per cpu = 200 cores
Memory	4 GB
Storage	320 GB
Network	Gigabit Ethernet (GbE) 4 Cisco-3750 GbE switches
OS	RedHat Enterprise Linux 5
	SCI Cluster
Processor	Intel Xeon DualCore 5060 3.2 GHz / 4 MB L2 cache / 1066 MHz 8 nodes x 2 cpus per node = 16 cpus 16 cpus x 2 cores per cpu = 32 cores
Memory	4 GB
Storage	320 GB
Network	Scalable Coherent Interface (SCI) Dolphin D334 card
OS	Linux CentOS 4.2

TABLE V
Benchmarked Resources

B. Computation Intensive Applications

Computation intensive applications can be characterized by a strong integer or float arithmetic, or by complex array manipulation. The Fourier Transformation (FT), Integer Sort (IS) and Embarrassingly Parallel (EP) kernels are such applications. In the remaining of this section we discuss results on up to 32 nodes. We have ran the benchmarks on 128 nodes (256 for EP) but the results were not different and are omitted here for space reasons.

1) Fourier Transformation Kernel (FT): It is a test for computation performance with a large memory footprint, solving differential equation using FFTs. This kernel also tests communication throughput by sending a few numbers of very large messages. For a class C problem with 16 workers, each worker sends 22 messages for a total amount of 180 MBytes. Notice that the original Fortran implementation uses some native operations on multi-dimensional arrays which are not available on Java. Thus, we have implemented some of these operations in Java, at a higher level, causing a large amount of integer operations through array indices computation.

If we take a look at the Fig.2, we see that the kernel could not start with a small number of nodes. While the MPI version ran from 2 nodes, we see that the Java versions only starts from 8 nodes, except for the Sun 1.5 version which was only able to start the kernel from 16 nodes. Actually, as this kernel deals with very large data structures, we encountered numerous *"OutOfMemory"* errors. Regarding the duration time, we can see that the ProActive version has about the same behaviour with 6 JVM out of 7. Compared to the MPI version, results

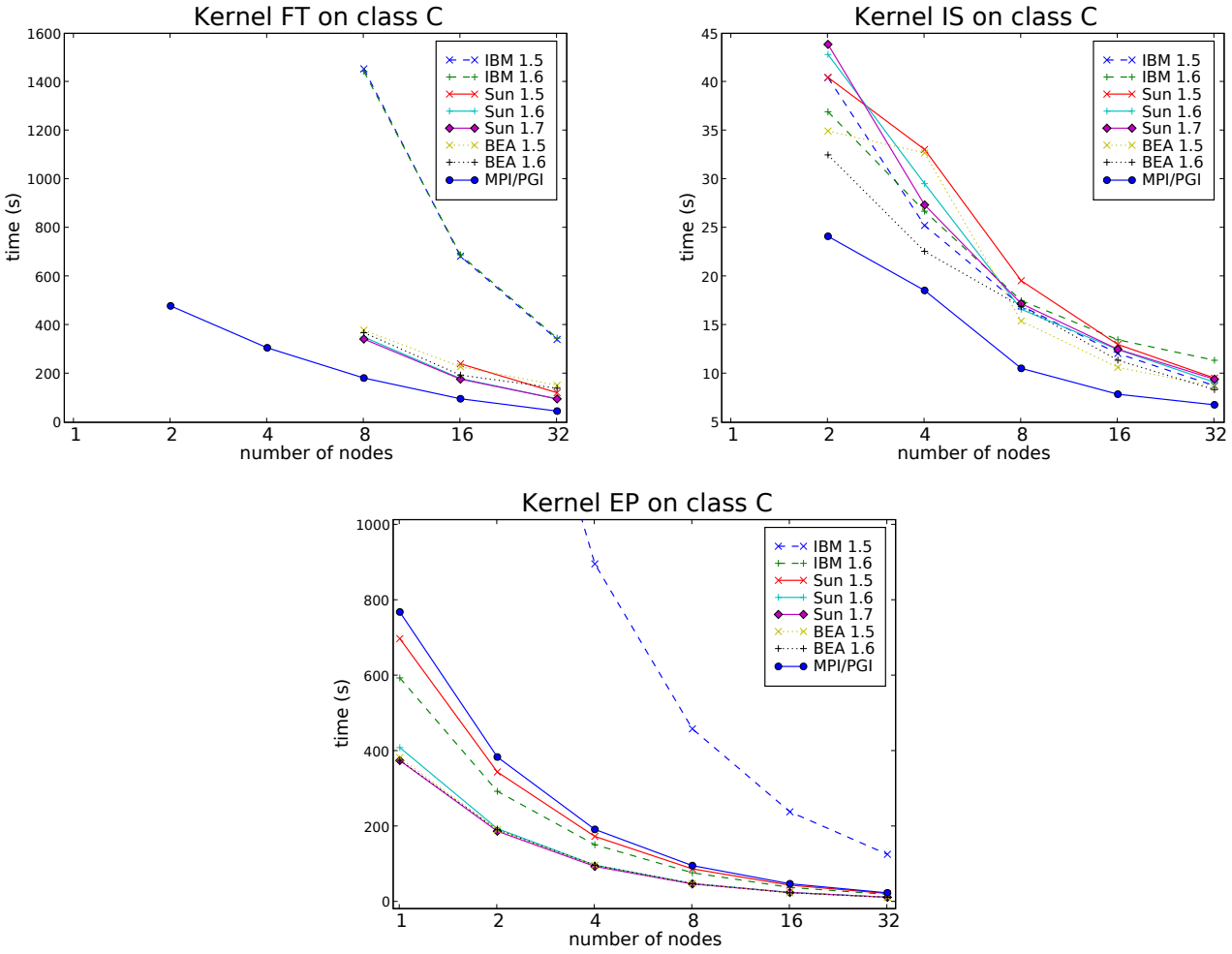

Fig. 2. Execution time of the computation intensive kernels (FT, IS and EP) for various JVMs on the Gigabit Ethernet cluster

are in the same order of magnitude.

2) Integer Sort Kernel (IS): It tests both computational speed and communication performance. It performs a bucket sort on a large array of integers (up to 550 MBytes for a class C problem). Thus, this kernel is mainly characterized by a large amount of data movements. On a class C problem with 16 workers, each worker sends to each other 65 messages for a total amount of 22 MBytes.

On the Fig.2, we see that all the JVM implementations have similar behaviours with an execution time which is not so far from the native MPI results (by a factor smaller than 2).

3) Embarrassingly Parallel Kernel (EP): It provides an estimation of the floating point performance by generating pseudo-random floating point values according to a Gaussian and uniform schemes. This kernel does not involve significant inter processor communication. Regarding the implementation in Java ProActive, some mathematical functions have been rewritten for performance issues with base 2 computation. This is the case with *pow* and *log* methods. A large amount of the operations involved in this kernel are some very simple operations such as bit shifting.

Figure 2 shows that the achievable floating point performance of Java is now quite competitive with native Fortran. With a problem class C, we can say that the overall behaviour of the various implementations of Java are the same, with a lack of performance for IBM 1.5. Furthermore, we note that for this kind of problem, the Java results are slightly better than the MPI ones.

C. Communication Intensive Applications

Communication intensive kernels are those which send a large amount of messages. The Conjugate Gradient (CG) and MultiGrid (MG) kernels are such applications.

1) Conjugate Gradient Kernel (CG): It is typical of unstructured grid computation. It is a strong test for communication speed and is highly dependent on the network latency. It deals with a very large amount of small messages (with a problem class C on 16 nodes, $429,248$ messages smaller than 50 bytes are sent) and a large amount of mid-size messages ($86,044$ messages of 300 KBytes are sent). When running a class C problem, CG kernel is composed of 75 iterations. In fact, this characterizes the unstructured communications aspect

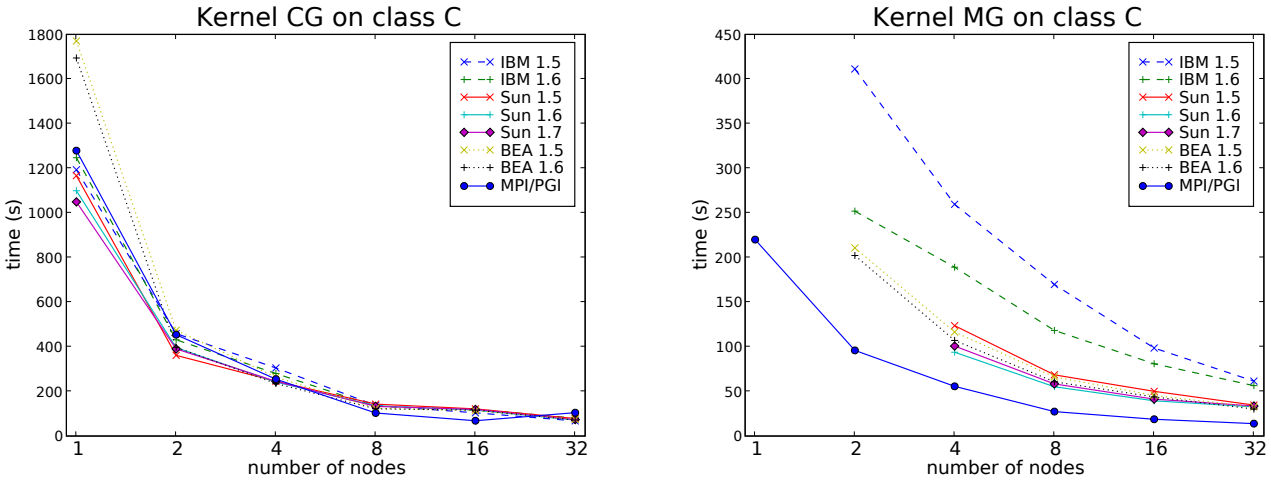

Fig. 3. Execution time of the communication intensive kernels (CG and MG) for various JVMs on the Gigabit Ethernet cluster

of this kernel.

Regarding performance comparison, Fig.3 shows the performance results on the Gigabit Ethernet cluster. We can see that almost all the JVM implementations (except BEA on 1 node) and native MPI version have about the same performance. Actually, in the Java ProActive implementation, CG kernel uses many of the *exchange* operators. Recall that it optimizes the synchronization between processes and eliminates unnecessary data duplications. It shows that to send a large number of messages of varying size ($429,248$ messages of less than 50 bytes and $86,044$ messages of 300 KBytes), the Java ProActive solution is as good as the native Fortran MPI solution. When looking at the performance comparison on the SCI cluster, presented on Fig.4, we see about the same behaviour as for the Gigabit Ethernet cluster. More precisely, the Fig.4(a) shows that MPI take a little more advantage of the low latency cluster, but not blatantly. If we now put more than 1 process per node, as the Fig.4(b) shows, we see that the achievable floating point performance increase significantly for MPI, but also for Java ProActive.

2) MultiGrid Kernel (MG): It is a simplified multi-grid problem. Topology is based on a vanilla hypercube (some edges are added to standard hypercube). It tests both short and long distance data communication with variable message size. When running with a problem class C on 16 nodes, a total of about $25,000$ messages are sent. Size distribution is as follows: 5000*1KB, 4032*2 KB, 4032*8KB, 4032*32KB, 4032*128KB and 4032*512KB. Also, MG deals with much larger data structures in memory than the CG kernel, causing memory problems.

Regarding performance comparison, Fig.3 shows the performance results on the Gigabit Ethernet cluster. Here, the important size of data structures, previously mentioned, is clearly visible. Indeed, when using only one node, the data structures are too large to be handled by the JVMs. To be able to perform a run, we need at least two nodes for the BEA and IBM JVMs, and 4 nodes for the Sun, with default garbage collector configuration. On the other hand, the native MPI version is able to run using only one node. Looking at the execution time, we see that Sun and BEA JVMs are twice as slow as the MPI version. The IBM JVM performance is even worse than other vendors VM . This lack of performance can be explained by the large amount of double and integer operations involved in.

When running on the SCI cluster, as shown on the Fig.4, we see that the MPI implementation takes a better advantage of the low latency cluster. When deploying one process per core (4 processes per node), as shown on the Fig.4(d), we obtain better results with the Java version, closing on the MPI performance.

VI. Conclusion and Future Work

In this paper we have reported on the design, implementation and benchmarks of a Java version of the NPBs using the ProActive middleware for distribution.

First we have shown that care is needed when writing HPC code in Java. The standard arithmetic methods have low performance compared to C equivalent. But when replacing them with an optimized version, it is possible to outperform equivalent native code. The memory overhead can be important when manipulating multi-dimensional arrays. This is easily addressed by using flattening techniques. Finally, avoiding premature optimization (such as inlining) helps the JIT and leads to better performance.

Second, we have compared the performance of a Java implementation of the NPBs to a Fortran MPI one (PGI 7.1). When considering strongly communicating applications, the speed and scalability of the Java ProActive implementation are, as of today still lower than MPI. On the MG and FT kernels, the overhead factor ranged from 1.5-2 on 16 nodes to 2-6 on 32 nodes. The lack of scalability in those benchmarks is mainly due to numerous small messages for which the ProActive overhead is significantly higher than the MPI one. We are working to reduce that overhead through size reduction

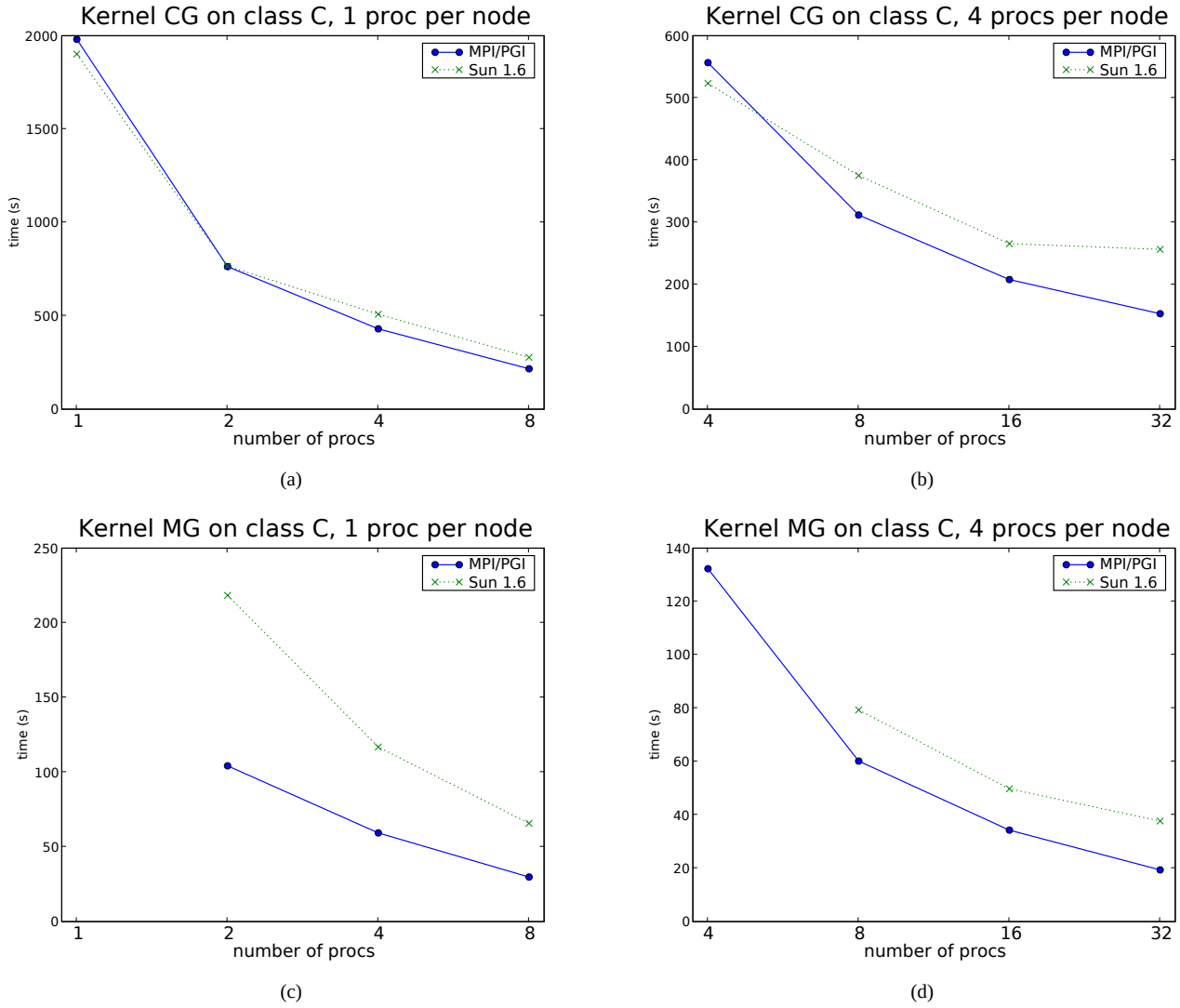

Fig. 4. Execution time of the communication intensive kernels (CG and MG) on the SCI cluster

of the message context but are dependent on the RMI layer. One solution would be to bypass RMI by defining a new protocol adapted to small messages.

On computational intensive benchmarks (IS and EP) the Java ProActive version performs is as effective as the Fortran MPI version on up to 64 machines.

We have also shown that it is possible to take advantage of high-performance interconnects (SCI) in a non-intrusive way. Using a network layer, JFS, it is possible to transparently use an SCI infrastructure without source code modification or reconfiguration. The differences between the Java implementation and the MPI one are narrower on these systems, showing the feasibility of this approach on high-performance interconnects. Moreover, the communication bottleneck can be further reduced using a mixed approach by putting several processes on a multi-core node to take advantage of local communications.

Overall, the results obtained are encouraging. We believe that the overhead of Java is acceptable when performing computational intensive tasks. Regarding communication intensive tasks, the lower performance can be partially overcome using mixed approach and optimized network layers. The HPC community has already worked on the issue and produced interesting results [19]. However, current JVM vendors have not developed efficient enough solutions yet.

ACKNOWLEDGMENTS

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

REFERENCES

[1] Bea systems jrockit. checklist and tuning guide for optimizing the bea jrockit jvm. http://www.oracle.com/technology/pub/articles/dev2arch/2007/12/jrockit-tuning.html.

[2] Sun microsystems. java se 6 hotspot[tm] virtual machine garbage collection tuning. http://java.sun.com/javase/technologies/hotspot/gc/gc_tuning_6.html.

[3] L. Baduel, F. Baude, and D. Caromel. Object-oriented spmd. In *Proceedings of Cluster Computing and Grid*, Cardiff, United Kingdom, may 2005.

[4] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, New York, NY, USA, Oct. 2006. ACM Press.

[6] D. Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[7] K. Datta, D. Bonachea, and K. Yelick. Titanium Performance and Potential: An NPB Experimental Study. *LECTURE NOTES IN COMPUTER SCIENCE*, 4339:200, 2006.

[8] M. Frumkin, H. Jin, and J. Yan. Implementation of NAS Parallel Benchmarks in High Performance Fortran. In *Proceedings of the 13th International Parallel Processing Symposium and the 10th Symposium on Parallel and Distributed Processing,(IPPS/SPDP'99), San Juan, Puerto Rico*, 1999.

[9] S. Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Trans. Program. Lang. Syst.*, 26(1):47–56, 2004.

[10] W. Huang, B. Abali, and D. Panda. A case for high performance computing with virtual machines. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 125–134. ACM Press New York, NY, USA, 2006.

[11] F. Huet, D. Caromel, and H. E. Bal. A high performance java middleware with a real application. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 2, Washington, DC, USA, 2004. IEEE Computer Society.

[12] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. *National Aeronautics and Space Administration (NASA), Technical Report NAS-99-011, Moffett Field, USA*, 1999.

[13] L. V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth, and G. Zheng. Programming Petascale Applications with Charm++ and AMPI. In D. Bader, editor, *Petascale Computing: Algorithms and Applications*, pages 421–441. Chapman & Hall / CRC Press, 2008.

[14] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the java hotspot™ client compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5(1):1–32, 2008.

[15] J. E. Moreira, S. P. Midkiff, and M. Gupta. A comparison of three approaches to language, compiler, and library support for multidimensional arrays in java. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 116–125, New York, NY, USA, 2001. ACM.

[16] M. Philippsen, R. F. Boisvert, V. Getov, R. Pozo, J. E. Moreira, D. Gannon, and G. Fox. Javagrande - high performance computing with java. In *PARA '00: Proceedings of the 5th International Workshop on Applied Parallel Computing, New Paradigms for HPC in Industry and Academia*, pages 20–36, London, UK, 2001. Springer-Verlag.

[17] S. Saini, J. Chang, R. Hood, and H. Jin. A Scalability Study of Columbia using the NAS Parallel Benchmarks. *Journal of Comput. Methods in Sci. and Engr*, 2006.

[18] G. L. Taboada, J. Touriño, and R. Doallo. Efficient java communication protocols on high-speed cluster interconnects. In *Proc. 31st IEEE Conf. on Local Computer Networks (LCN'06)*, pages 264–271, Tampa, FL, 2006.

[19] R. V. van 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, June 2005.

[20] T. Würthinger, C. Wimmer, and H. Mössenböck. Array bounds check elimination for the java hotspot™ client compiler. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 125–133, New York, NY, USA, 2007. ACM.