NPB-MPJ: NAS Parallel Benchmarks Implementation for Message-Passing in Java

Damián A. Mallón, Guillermo L. Taboada, Juan Touriño, and Ramón Doallo

Computer Architecture Group
Dept. of Electronics and Systems, University of A Coruña
A Coruña, 15071 Spain
Email: {dalvarezm,taboada,juan,doallo}@udc.es

Abstract—**Java is a valuable and emerging alternative for the development of parallel applications, thanks to the availability of several Java message-passing libraries and its full multithreading support. The combination of both shared and distributed memory programming is an interesting option for parallel programming multi-core systems. However, the concerns about Java performance are hindering its adoption in this field, although it is difficult to evaluate accurately its performance due to the lack of standard benchmarks in Java.**

This paper presents NPB-MPJ, the first extensive implementation of the NAS Parallel Benchmarks (NPB), the standard parallel benchmark suite, for Message-Passing in Java (MPJ) libraries. Together with the design and implementation details of NPB-MPJ, this paper gathers several optimization techniques that can serve as a guide for the development of more efficient Java applications for High Performance Computing (HPC). NPB-MPJ has been used in the performance evaluation of Java against C/Fortran parallel libraries on two representative multi-core clusters. Thus, NPB-MPJ provides an up-to-date snapshot of MPJ performance, whose comparative analysis of current Java and native parallel solutions confirms that MPJ is an alternative for parallel programming multi-core systems.

I. Introduction

Java has an important presence in the industry and academia due to its appealing characteristics: built-in networking and multithreading support, object orientation, platform independence, portability, security, it is the main training language for computer science students and has a wide community of developers. Moreover, the gap between native languages, such as C/C++ and Fortran, and Java performance has been narrowing for the last years, thanks to the Java Virtual Machine (JVM) Just-in-Time (JIT) compiler that obtains near native performance from Java bytecode. Therefore, nowadays Java is a competitive alternative in HPC.

The clusters, due to their scalability, flexibility and acceptable ratio performance/cost, have an important presence in HPC. Currently, multi-core clusters are the most popular option for the deployment of HPC infrastructures. These systems are usually programmed with native languages using message-passing libraries, especially MPI [1], which are targeted to distributed memory systems. However, the hybrid architecture (shared/distributed memory) of the multi-core systems demands the use of hybrid programming approaches, such as the use of MPI+OpenMP, in order to take advantage of the available processing power. An interesting alternative is the use of Java for parallel programming multi-core systems. In fact, the Java built-in networking and multithreading support makes this language especially suitable for this task.

Moreover, the availability of different Java parallel programming libraries, such as Message-Passing in Java (MPJ) libraries and ProActive [2] [3], an RMI-based middleware for multithreaded and distributed computing focused on Grid applications, eases Java's adoption. In this scenario, a comparative evaluation of Java for parallel computing against native solutions is required in order to assess its benefits and disadvantages. Thus, we provide an extensive implementation of the standard parallel benchmark suite, the NAS Parallel Benchmarks (NPB) [4], for MPJ. Moreover, as JVM technology and MPJ libraries are actively evolving, this paper also aims to be an up-to-date snapshot of MPJ performance, compared to existing Java and native solutions for parallel programming.

The structure of this paper is as follows: Section 2 presents background information about MPJ. Section 3 introduces the related work in Java NPB implementations. Section 4 describes the design, implementation and optimization of NPB-MPJ, our NPB implementation for MPJ. Comprehensive benchmark results from an NPB-MPJ evaluation on two representative multi-core clusters are shown in Section 5. Moreover, additional NPB results from different Java and native parallel libraries (Java threads, ProActive, MPI and OpenMP) are also shown for comparison purposes. Finally, Section 6 concludes the paper.

II. Message-Passing in Java

The message-passing is the most widely used parallel programming model as it is portable, scalable and usually provides good performance. It is the preferred choice for parallel programming distributed memory systems such as clusters, which provide higher scalability and performance than shared memory systems. Regarding native languages, 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 [5]). Most of them have developed their own MPI-like binding for the Java language. The two main

proposed APIs are the mpiJava API [6] and MPJ API [7], whose main differences lay on naming conventions of variables and methods. The most relevant MPJ libraries are next shown.

The mpiJava [8] library consists of a collection of wrapper classes that call a native MPI implementation (e.g., MPICH or OpenMPI) through Java Native Interface (JNI). This wrapper-based approach provides efficient communication relying on native libraries, adding a reduced JNI overhead. However, mpiJava currently only supports some combinations of JVMs and MPI libraries, 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.

The main mpiJava drawbacks are solved with the use of "pure" Java (100% Java) message-passing libraries, that implement the whole messaging system in Java. However, these implementations are usually less efficient than mpiJava. MPJ Express [9] is a thread-safe and "pure" MPJ library that implements the mpiJava API. Furthermore, it provides Myrinet support (through the native Myrinet eXpress, MX, communication library). MPJ/Ibis [10] is another "pure" MPJ library integrated in the Ibis parallel and distributed Java computing framework [11]. MPJ/Ibis implements the MPJ API and also provides Myrinet support, although on GM, which shows poorer performance than MX.

Additionally, there are several recent MPJ projects, such as Parallel Java [12], Jcluster [13] and P2P-MPI [14], developments tailored to hybrid, heterogeneous and grid computing systems, respectively. Previous Java message-passing libraries, although raised many expectations in the past, are currently out-of-date and their interest is quite limited. This important number of past and present projects is the result of the sustained interest in the use of Java for parallel computing.

III. NAS PARALLEL BENCHMARKS IMPLEMENTATIONS IN JAVA

The NAS Parallel Benchmarks (NPB) [4] consist of a set of kernels and pseudo-applications taken primarily from Computational Fluid Dynamics (CFD) applications. These benchmarks reflect different kinds of computation and communication patterns that are important across a wide range of applications. Therefore, they are the de facto standard in parallel performance benchmarking. There are NPB implementations for the main parallel programming languages and libraries, such as MPI (from now on NPB-MPI), OpenMP (from now on NPB-OMP), High Performance Fortran (HPF), UPC, and Co-Array Fortran.

Regarding Java, currently there are three NPB implementations, apart from NPB-MPJ, namely the multithreaded [4] (from now on NPB-JAV), the ProActive [15] (from now on NPB-PA), and the Titanium [16] implementations. However, these three developments present several drawbacks in order to evaluate the capabilities of Java for parallel computing.

NPB-JAV is limited to shared memory systems and thus its scalability is lower than the provided by the distributed memory programming model. Regarding NPB-PA, although relies on a distributed memory programming model, the use of an inefficient communication middleware such as RMI limits its performance scalability. Titanium is an explicitly parallel dialect of Java, so its portability is quite limited. Moreover, the NPB-MPJ development is of great interest as the reference implementation of the NPB is written in MPI. Thus, Java can be evaluated within the target programming model of the NPB, the message-passing paradigm.

Another motivation for the implementation of the NPB-MPJ is the current lack of parallel benchmarks in Java. The most noticeable related project is the Java Grande Forum (JGF) benchmark suite [17] that consists of: (1) sequential benchmarks, (2) multithreaded codes, (3) MPJ benchmarks, and (4) the language comparison version, which is a subset of the sequential benchmarks translated into C. However, the JGF benchmark suite does not provide with the MPI counterparts of the MPJ codes, allowing only the comparison among MPJ libraries and Java threads. Moreover, its codes are less representative of HPC kernels and applications than the NPB kernels and applications.

Therefore, NPB-MPJ is a highly interesting option to implement a parallel benchmark suite in Java. The use of MPJ allows the comparative analysis of the existing MPJ libraries and the comparison between Java and native message-passing performance, using NPB-MPJ and NPB-MPI, respectively. Moreover, it also serves to evaluate Java parallel libraries which have implemented the NPB, such as ProActive (NPB-PA).

Previous efforts in the implementation of NPB for MPJ have been associated with the development of MPJ libraries. Thus, JavaMPI [18] included EP and IS kernels, the two shortest NPB codes. Then, the CG kernel was implemented for MPJava [19]. Finally, P2P-MPI [14] also implemented the EP and IS kernels.

NPB-MPJ enhances these previous efforts implementing an extensive number of benchmarks: the CG, EP, FT, IS, MG and DT kernels and the SP pseudo-application. An approximate idea of the implementation effort carried out in NPB-MPJ can be estimated using the SLOC (Source Lines Of Code) metric: CG has 1000 SLOC, EP 350, FT 1700, IS 700, MG 2000, DT 1000, and SP 4300 SLOC. NPB-MPJ has as a whole approximately 11.000 SLOC. Moreover, NPB-MPJ uses the most extended Java message-passing API, the mpiJava API (mpiJava and MPJ Express). Finally, it provides support for automating the benchmarks execution and the graphs and performance reports generation. NPB-MPJ has significantly increased the availability of standard Java parallel benchmarks.

IV. DESIGN, IMPLEMENTATION AND OPTIMIZATION OF NPB-MPJ

NPB-MPJ is the implementation of the standard NPB benchmark suite for MPJ performance evaluation. This suite allows: (1) the comparison among MPJ implementations; (2)

the evaluation of MPJ against other Java parallel libraries; (3) the assessment of MPJ versus MPI; and finally, (4) it provides an example of best programming practices for performance in Java parallel applications. This section presents the design of NPB-MPJ, the implementation of its initial version and its subsequent performance optimization.

A. NPB-MPJ Design

The NPB-MPJ development has been based on the NPB-MPI implementation, which consists of Fortran MPI codes except for IS and DT, which are C MPI kernels. The use of the message-passing programming model determines that NPB-MPJ and NPB-MPI share several characteristics, and thus NPB-MPJ has followed the NPB-MPI SPMD paradigm, its workload distribution and communications. The NPB-JAV implementation has also served as source for NPB-MPJ. Although its master-slave paradigm has not been selected for NPB-MPJ, its Java-specific solutions, such as its complex numbers support or its timing methods, have been useful for NPB-MPJ.

An important issue tackled in NPB-MPJ has been the selection between a "pure" object oriented design or an imperative approach through the use of "plain objects". In order to maximize NPB-MPJ performance, it has been opted for the "plain objects" design as it reduces the overhead of the "pure" object oriented design (up to a 95%). The overhead derived from an intensive use of object orientation in numerical codes has been recognized as significant in the related literature [20]. An example of this design decision is the complex numbers support in NPB-MPJ. As Java does not have a complex number primitive datatype and the NPB uses them thoroughly, NPB-MPJ has implemented its own support. Thus, a complex number is implemented as a two-element array (real and imaginary parts). This approach presents less overhead than the implementation of complex number objects, which trades off a clear design and the encapsulation features for higher access overhead, especially when dealing with arrays of complex number objects.

B. NPB-MPJ Implementation

NPB-MPJ consists of the CG, EP, FT, IS, MG and DT kernels and the SP pseudo-application. A brief description of the implemented benchmarks is next presented. The CG kernel is a sparse iterative solver that tests communications performance in sparse matrix-vector multiplications. The EP kernel is an embarrassingly parallel code that assesses the floating point performance, without significant communications. The FT kernel performs a series of 3-D FFTs on a 3-D mesh that tests aggregated communication performance. The IS kernel is a large integer sort that evaluates both integer computation performance and the aggregated communication throughput. MG is a simplified multigrid kernel that performs both short and long distance communications. The DT (Data Traffic) kernel operates with graphs and evaluates communication throughput. The SP (Scalar Pentadiagonal) pseudo-application

is a simulated CFD application. This wide range of implemented benchmarks assures a broad performance analysis. Each kernel has presented its particular implementation issues. Moreover, there is a common issue in NPB-MPJ codes, which is the data storage handling, as next presented.

1) Java Array Handling: The NPB handles arrays of up to five dimensions. In native languages it is possible to define multidimensional arrays whose memory space is contiguous, unlike Java, where an n-dimensional array is defined as an array of $n-1$ dimensional arrays. The main issue for NPB-MPJ is the lack of support for the direct send of logically contiguous elements in multidimensional arrays (e.g., two rows from a C two-dimensional array). In MPI it is possible to communicate adjacent memory regions. In MPJ this has to be done through multiple communication calls or buffering the data in a one dimensional array in order to perform a single communication. The latter is the option initially implemented in NPB-MPJ, trying to minimize the communication overhead. However, this technique shows an important buffering overhead, which has motivated the search for a more efficient alternative.

The solution is the array flattening, which consists of the mapping of a multidimensional array in a one dimensional array. Thus, NPB-MPJ only uses one dimensional arrays. In order to reference a concrete element it is required a positioning method that maps an n-dimensional location into its corresponding one dimensional position. NPB-MPJ has implemented this mapping function so that adjacent elements in the C/Fortran versions are contiguous in Java, in order to provide an efficient access to the data. A particular application of the array flattening in NPB-MPJ has been applied in the complex number arrays, replacing the two dimensional array ($complexNum\_arr[2][N]$) for a one dimensional array ($complexNum\_arr[2xN]$). In this case, in order to exploit the locality of the data the positioning method maps a complex number to contiguous positions ($complexNum\_arr[x]$ and $complexNum\_arr[x+1]$). Therefore, it is direct the complex numbers support in MPJ communications. The array flattening has yielded a significant performance increase, not only in avoiding data buffering and reducing the number of communications calls, but also on accessing the array elements.

C. NPB-MPJ Optimization

Once a fully functional NPB-MPJ first implementation has been developed, several optimizations have been applied to the benchmark codes. This subsection presents them.

1) JVM JIT Compiler-based Optimization: The Java bytecode can be either interpreted or compiled for its execution by the JVM, depending on the number of times the method to which the bytecode belongs is invoked. As the bytecode compilation is an expensive operation that increases significantly the runtime, it is reserved to heavily-used methods. However, at JVM start-up is not always possible to find out these most heavily-used methods. Therefore, the JVM gathers at runtime information about methods invocation and their computational cost, in order to guide the compiler optimization of the JVM JIT compiler. The JIT compiler compiles Java

bytecode to native code or recompiles native code applying further optimizations in order to minimize the overall runtime of a Java application. Its operation is guided by the profiling information of the executed methods and the JVM policy.

Thus, regarding JIT compiler operation, two paradoxes occur in Java applications, and in particular in NPB-MPJ: (1) an optimized code yields worse performance than an unoptimized code and (2) a code with many invocations to simple methods runs faster than a code with all the methods inlined. In the first case, the JIT compiler optimizes more aggressively the methods that fall beyond certain load threshold. In NPB-MPJ the manual code optimization of some methods resulted in initially lower execution time than the first invocation of the unoptimized methods. Therefore, the JIT compiler does not optimize aggressively their code and eventually the overall execution time is slower than the previous version. In the latter case, a Java application with multiple simple methods that are constantly invoked run faster than a Java code with less methods and whose method invocations are inlined. The simple methods are more easily optimized, in terms of compilation time and in quality of the generated code. Moreover, the JVM gathers more runtime information of methods constantly invoked, allowing a more effective optimization of the target bytecode.

NPB-MPJ takes advantage of the JIT compiler operation. Thus, in general the code has not been manually optimized, relying on the JIT compiler for this task. However, there are a few exceptions such as the use in the most inner loops of bit shifting operations instead of integer multiplication, divisions by powers of two and the optimization of complex numbers operations in the FT kernel. Another exception is the use of the relative positioning. Instead of accessing to contiguous elements every time through global positioning method calls, it is used the location of the first element as base position (loop invariant) and then adjacent elements are accessed with their corresponding offsets to the base position. This optimization is only applied in the most inner loop.

Moreover, the benchmarks code has been refactored towards simpler and independent methods. More concretely, simple methods for the multiplication and division of complex numbers, and for mapping elements from multidimensional to one dimensional arrays have been implemented, rather than inlining these operations in the code in order to avoid the method invocation overhead. The performance improvement for NPB-MPJ with this optimization has been quite significant, especially for the SP pseudo-application. Furthermore, the performance optimization techniques used in this work are easily applicable to other codes, whose performance can be greatly improved as it has been done for the SP pseudo-application, for which up to a 2800% performance increase has been reported.

V. Performance Evaluation

A. Experimental Configuration

An evaluation of Java for parallel programming using NPB-MPJ has been carried out on two multi-core clusters. The first one is a Gigabit-Ethernet cluster that consists of 8 nodes, each with 2 dual-core processors (Intel Xeon 5060 at 3.2 GHz) and 4 GB of memory. The OS is Linux CentOS 4.4 with C compiler gcc 4.1.2. The JVM used is Sun JDK 1.6.0_02. The native MPI library is MPICH2 1.0.7 with the SSM (Sockets and Shared Memory) channel, which uses sockets for internode transfers and shared memory for intranode communication. This system results have been obtained using 1 core per node, except for 16 and 32 processes, for which 2 and 4 cores per node, respectively, have been used.

The second system is an Infiniband cluster that consists of 4 HP Integrity rx7640 nodes, each of them with 8 Montvale Itanium2 (IA64) dual-core processors at 1.6 GHz and 128 GB of memory. The Infiniband NIC is a dual 4X IB port (16 Gbps of theoretical effective bandwidth). The native low-level communication middleware used by Java is SDP Sockets Direct Protocol (SDP), whereas MPI uses the Infiniband Verbs driver from the Open Fabrics Enterprise Distribution (OFED) 1.2. The OS is SUSE Linux Enterprise Server 10 with C compiler Intel icc 9 (with OpenMP support). The JVM is BEA JRockit 5.0 (R27.6), the only JVM 1.5 or higher (prerequisite for the evaluated MPJ libraries) available for Linux IA64 at the evaluation time. Regarding this point, Sun has recently released its JVM 1.6u7 for Linux IA64, but preliminary tests have shown its poorer NPB-MPJ performance than the JRockit 5.0, so the Sun JVM has not been included in the current work. The native MPI library is HP MPI 2.2.5.1 that uses the Infiniband Verbs driver for internode communication and shared memory for intranode communication. This system results shown have been obtained using up to 8 cores per node. Thus, the number of nodes used is $\lceil ncores/8 \rceil$, where $ncores$ is the number of cores used.

The evaluated MPJ libraries are MPJ Express 0.27 and mpiJava 1.2.5x. It has been used the NPB-MPI/NPB-OMP version 3.0. The ProActive version used is the 3.2.

The performance results considered in this work have been derived from the sample of several iterations of the kernel/application solver method, ignoring the initialization times and the previous warm-up iterations. The metric that has been considered is MOPS (Millions of Operations Per Second), which measures the kernel operations, that differ from the CPU operations issued.

B. NPB Kernels Performance on the Gigabit Ethernet cluster

Figures 1 and 2 show NPB-MPI, NPB-MPJ and NPB-PA kernels performance on the Gigabit Ethernet multi-core cluster. The NPB-MPJ results have been obtained using two MPJ libraries, MPJ Express and mpiJava, in order to compare them. The results are shown in MOPS, in order to compare the absolute performance of the different parallel libraries. Moreover, it has been used Classes A and B for problem sizes as these are small workloads targeted for execution on a reduced number of cores.

Regarding CG kernel performance, the evaluated implementations show poor scalability using more than 1 core per node. In this scenario NPB-PA obtains the lowest results, whereas
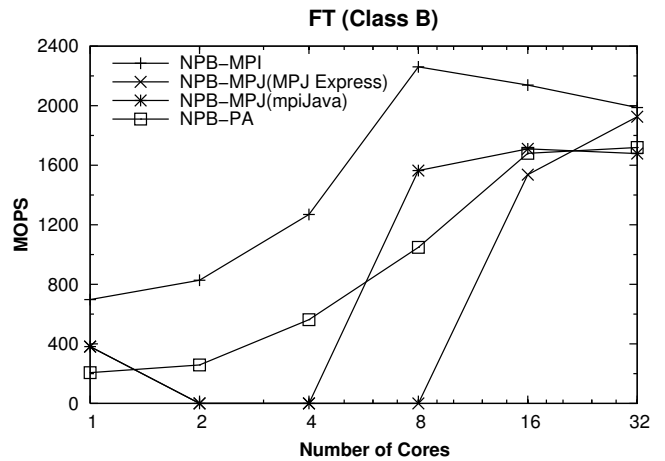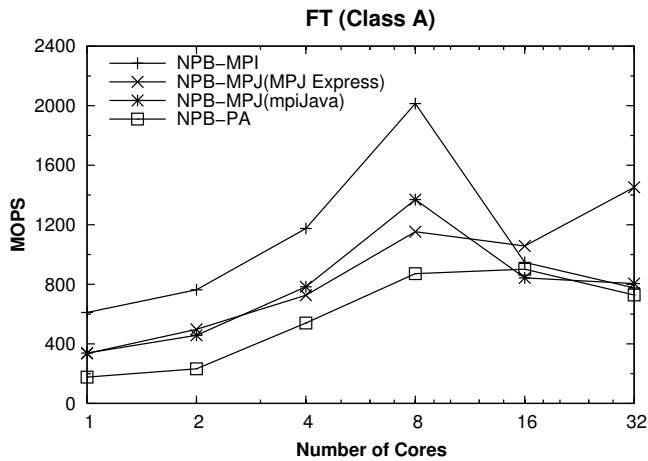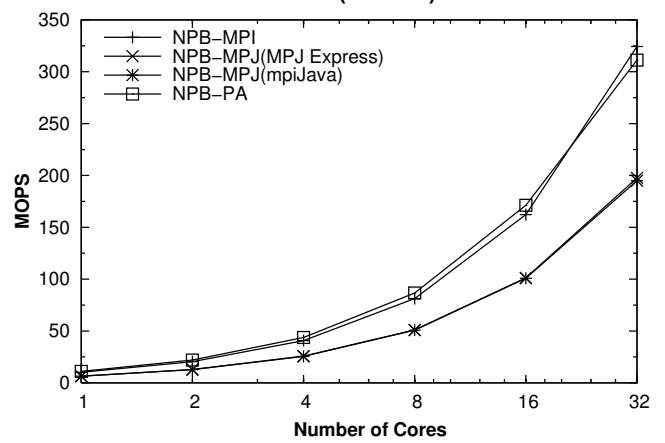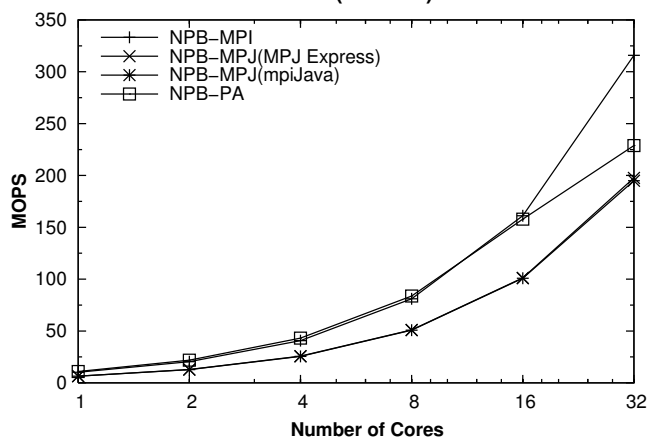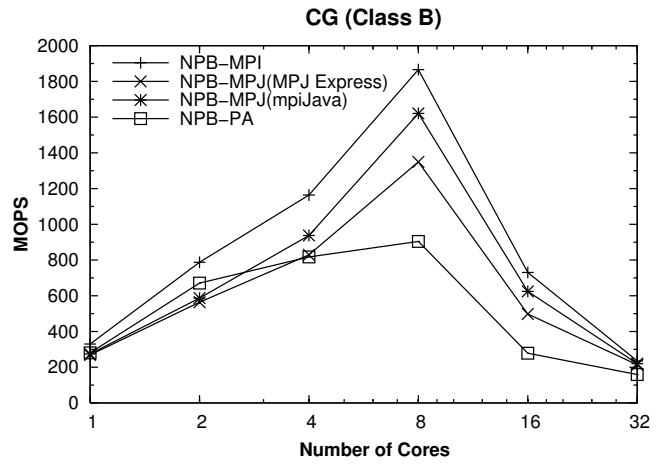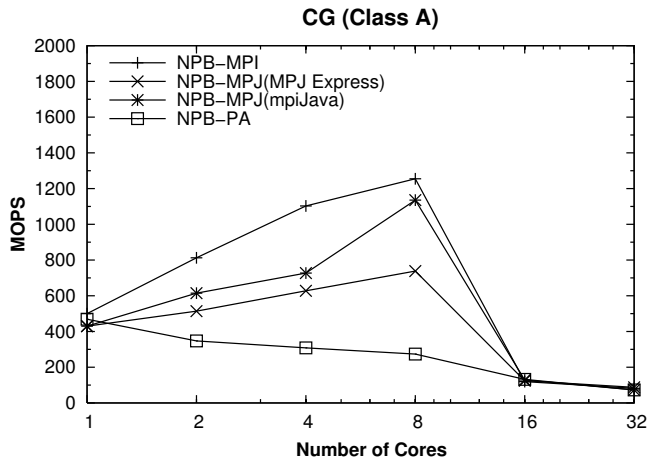
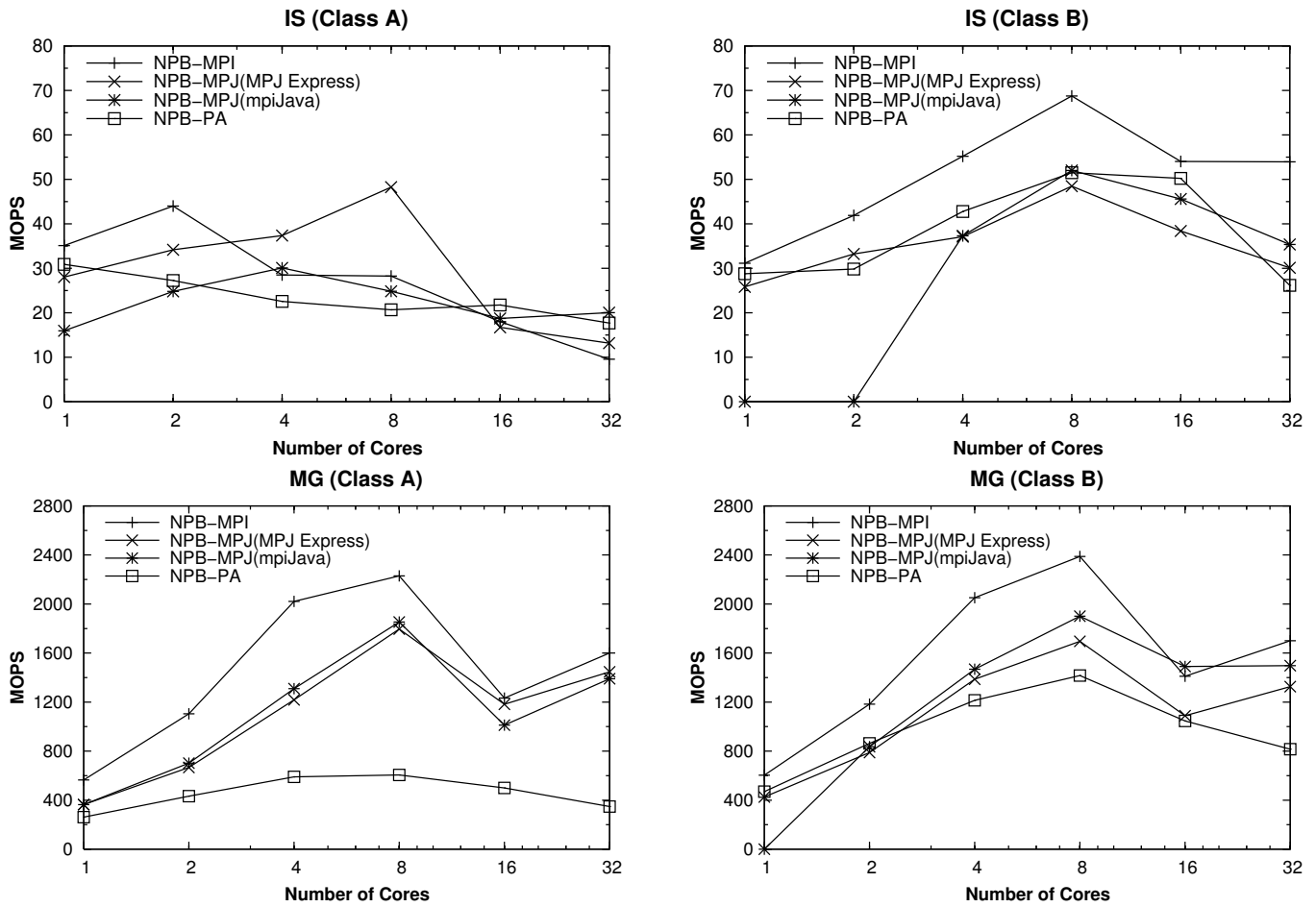Figure 1.   CG, EP and FT kernels performance on the Gigabit Ethernet multi-core cluster

Figure 2. IS and MG kernels performance on the Gigabit Ethernet multi-core cluster

the performance of NPB-MPJ is good, achieving with mpiJava up to a 86% of the results of NPB-MPI (Class B, 8 cores). Using MPJ Express, NPB-MPJ obtains up to a 72% of NPB-MPI performance.

The EP kernel is an "embarrassingly parallel" benchmark, with few communications that have low impact on the overall performance. Therefore, its speedup is almost lineal. Thus, NPB-MPJ achieves up to a 62% of the performance of NPB-MPI (Class B, 32 cores). Finally, the high results of NPB-PA, quite similar to NPB-MPI, are especially remarkable.

Within FT kernel results, the NPB-MPI shows the best performance, although for Class A and 16-32 cores is lower than NPB-MPJ (up to a 47%). In this case, the intensive use of complex numbers in this kernel has not prevented NPB-MPJ from showing relatively high performance (for Class B and 32 cores NPB-MPJ obtains the 96% of NPB-MPI results). However, the memory requirements of FT Class B workload have made it impossible to run this kernel with NPB-MPI on less than 8-16 cores, with mpiJava and MPJ Express, respectively. In this scenario, NPB-PA has shown its robustness running in all circumstances and showing competitive performance, quite similar to mpiJava and slightly lower than MPJ Express.

Regarding IS graphs, NPB-MPJ usually shows the highest performance for Class A, whereas NPB-MPI obtains the best results for Class B. The scalability of IS is quite limited for message-passing libraries as it is a communication intensive kernel.

Finally, the best MG performance has been obtained using NPB-MPI, whereas NPB-PA shows generally the worse results. NPB-MPJ obtains quite similar performance for the two MPJ libraries, mpiJava and MPJ Express, which are half-way between NPB-MPI and NPB-PA results. However, NPB-MPJ shows the best scalability, as the larger the number of cores, the closer to NPB-MPI performance. In fact, NPB-MPJ achieves around 80-90% of NPB-MPI performance on 32 cores.

The analysis of these results can also be presented in terms of the four main evaluations that can be performed with NPB-MPJ (see Section 4). The first is the comparison among MPJ implementations. The performance differences between mpiJava and MPJ Express are mainly explained by their communication efficiency, which is usually higher for mpiJava as it relies on a native MPI library (in our testbed on MPICH2) rather than on "pure" Java communications. CG and MG results clearly confirm this, where mpiJava outperforms MPJ

Express. However, on FT and IS MPJ Express outperforms mpiJava on some cases. EP results are almost equal for both libraries as this kernel performs few communications. Nevertheless, mpiJava drawbacks can also be seen, as it could not be run in some cases for FT, IS and MG, always with the Class B problem size. The reason is the instability of the JVM, compromised with the access to the native MPI code through JNI. The configurations where MPJ Express results are not shown are due to JVM heap size problems, as MPJ Express demands more resources than mpiJava.

The second evaluation that can be performed with Figures 1 and 2 results is the comparison of MPJ against other Java parallel libraries, in this case ProActive. ProActive is an RMI-based middleware, and for this reason its performance is usually lower than MPJ libraries, whose communications are based on MPI or on Java sockets. In fact, the results show that the scalability of NPB-PA is worse than the NPB-MPJ one. However, ProActive is a middleware more robust and stable than the MPJ libraries as it could execute all the benchmarks, without instability or resource exhaustion issues.

The third analysis that has been done is the comparison of MPJ against native MPI. The presented results show that MPI generally outperforms MPJ, around a mean 25%. However, MPJ outperforms MPI in some scenarios, such as on IS and FT Class A, although for cases where MPI scalability was very poor.

Finally, the analysis of NPB-MPJ results has shown that MPJ does not especially benefit from the use of more than a core per node, except for EP. The best MPJ performance has been usually obtained on 8 cores (1 core per node). This behavior is not exclusive of MPJ, as MPI presents similar results. The reason is that the workloads considered (Class A and B) are relatively small and the impact on performance of the Gigabit Ethernet high start-up latency is important. Moreover, the network contention also impacts throughput. Thus, although a message-passing library can take advantage of shared memory transfers, a Gigabit Ethernet network constitutes the main performance bottleneck, especially when several processes are scheduled per cluster node.

C. Performance of NPB-MPJ Kernels on the Infiniband cluster

Figure 3-4 shows NPB-MPI, NPB-MPJ, NPB-OMP and NPB-JAV performance on the Infiniband multi-core cluster. The NPB-MPJ results have been obtained using only MPJ Express as the JRockit JVM + HP MPI + Linux IA64 combination is not supported by mpiJava. This is an example of the mpiJava lack of portability drawback. NPB-PA results are not shown for clarity purposes due to its low performance. The results are shown using speedups as both shared (NPB-OMP and NPB-JAV) and distributed (NPB-MPI and NPB-MPJ) memory programming models have been evaluated. Thus, it is required a performance metric that does not depend on the particular experimental results. Thus, the speedup metric has been selected due to its usefulness in order to analyze the scalability of parallel codes.

Regarding CG results, they are quite dependent on the problem workload. Thus, NPB-MPJ speedups are small for Class A and high for Class B. NPB-JAV shows generally the best scalability, although it is limited to shared memory scenarios.

The EP kernel, due to its small communications, shows again a high parallel efficiency on this cluster.

Regarding FT performance, the shared memory implementations, NPB-JAV and NPB-OMP, obtain the best speedups. For this kernel the message-passing libraries show smaller scalability, especially for NPB-MPJ with Class A workload. Nevertheless, the gap between NPB-MPI and NPB-MPJ narrows for Class B, even obtaining similar results (up to 8 cores).

IS is a communication intensive kernel whose NPB-OMP implementation presents important slowdowns, as well as the NPB-MPJ one, which shows significantly low speedups. The best performance is obtained by NPB-MPI, followed by NPB-JAV.

Finally, MG kernel results are quite similar among them, especially for Class B workload. The most significant results are that NPB-MPI outperforms NPB-MPJ from 4 and 16 cores, for Class A and Class B, respectively, and that NPB-JAV shows better results than NPB-OMP on 4-8 cores.

Additionally, the results shown in Figures 3 and 4 allow the evaluation of MPJ against Java threads and the comparison of the scalability of Java versus native parallel libraries (MPI and OpenMP).

Regarding the first evaluation, NPB-JAV usually outperforms NPB-MPJ, and even outperforms the native results on 8 cores for some configurations. Nevertheless, NPB-JAV scalability is limited to shared memory systems, whereas NPB-MPJ takes advantage of the higher number of cores of the whole distributed memory system to achieve higher speedups (especially for Class B, except IS), quite similar to NPB-MPI scalability. In this scenario is clear that MPJ requires significant workloads in order to scale performance, as small problem sizes (Class A) impact negatively on their results.

D. Performance of SP Pseudo-application

This subsection presents SP pseudo-application performance on the Gigabit Ethernet (Figure 5) and Infiniband (Figure 6) multi-core clusters. The graphs are presented using the same layout of the previous graphs shown for each system. Thus, SP results are presented using MOPS and speedups on the Gigabit Ethernet and Infiniband clusters, respectively. NPB-PA does not include SP code. SP requires a square number of processes (1,4,9,16,25), and it has been used on the Gigabit Ethernet cluster 1 core per node, except for 16 and 25 cores, where 2 and 3 cores per node have been used.

The results shown confirm the analysis presented in the previous subsections. Thus, NPB-MPJ results are quite close to NPB-MPI performance, even outperforming the native results on the Gigabit Ethernet cluster. In this scenario the scalability of MPJ is higher than for MPI, as for 1 core MPI doubles MPJ performance and on 25 cores MPJ either outperforms (Class A) or obtains slightly lower performance, around 10%, (Class
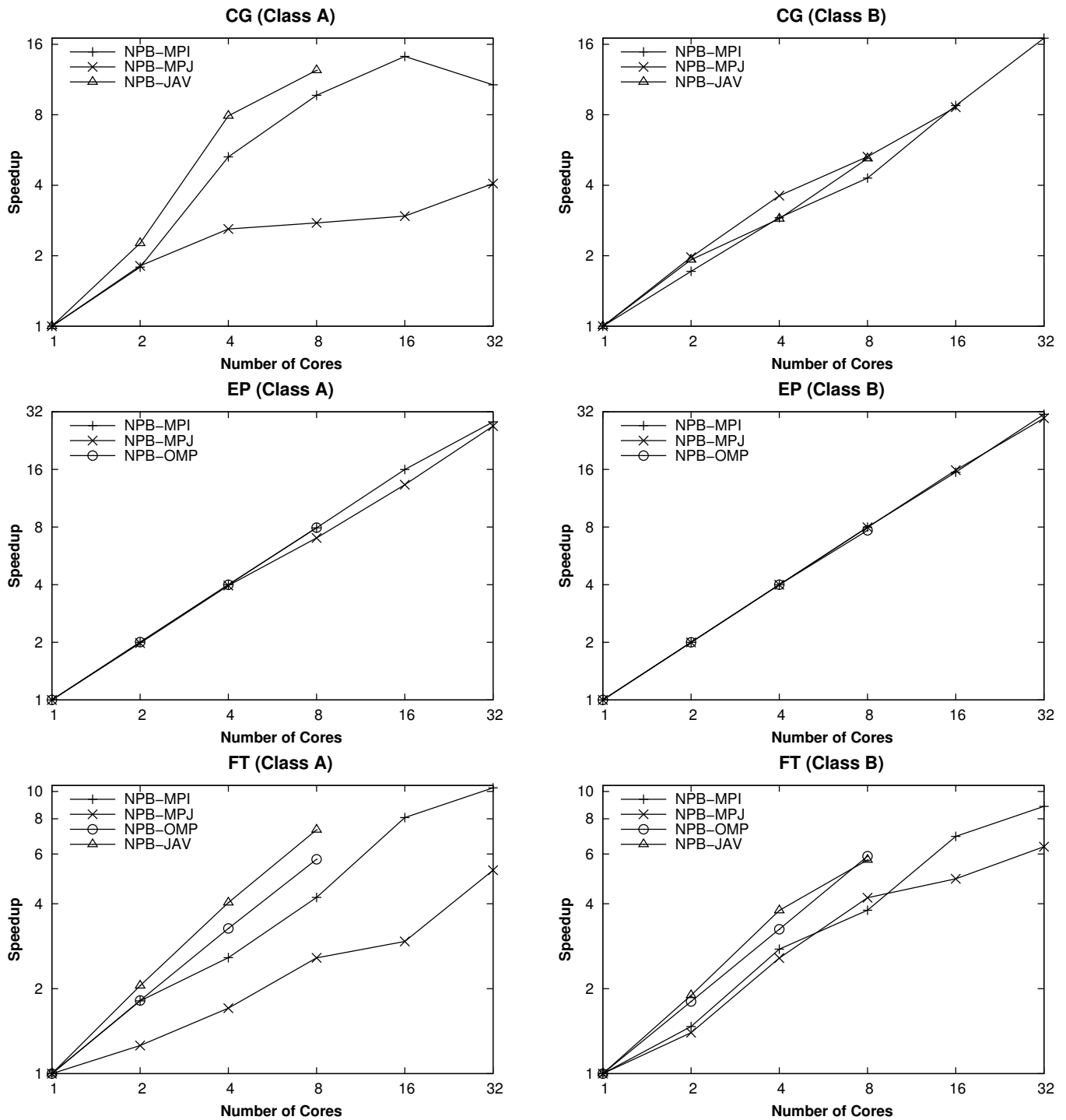
Figure 3.   CG, EP and FT kernels performance on the Infiniband multi-core cluster
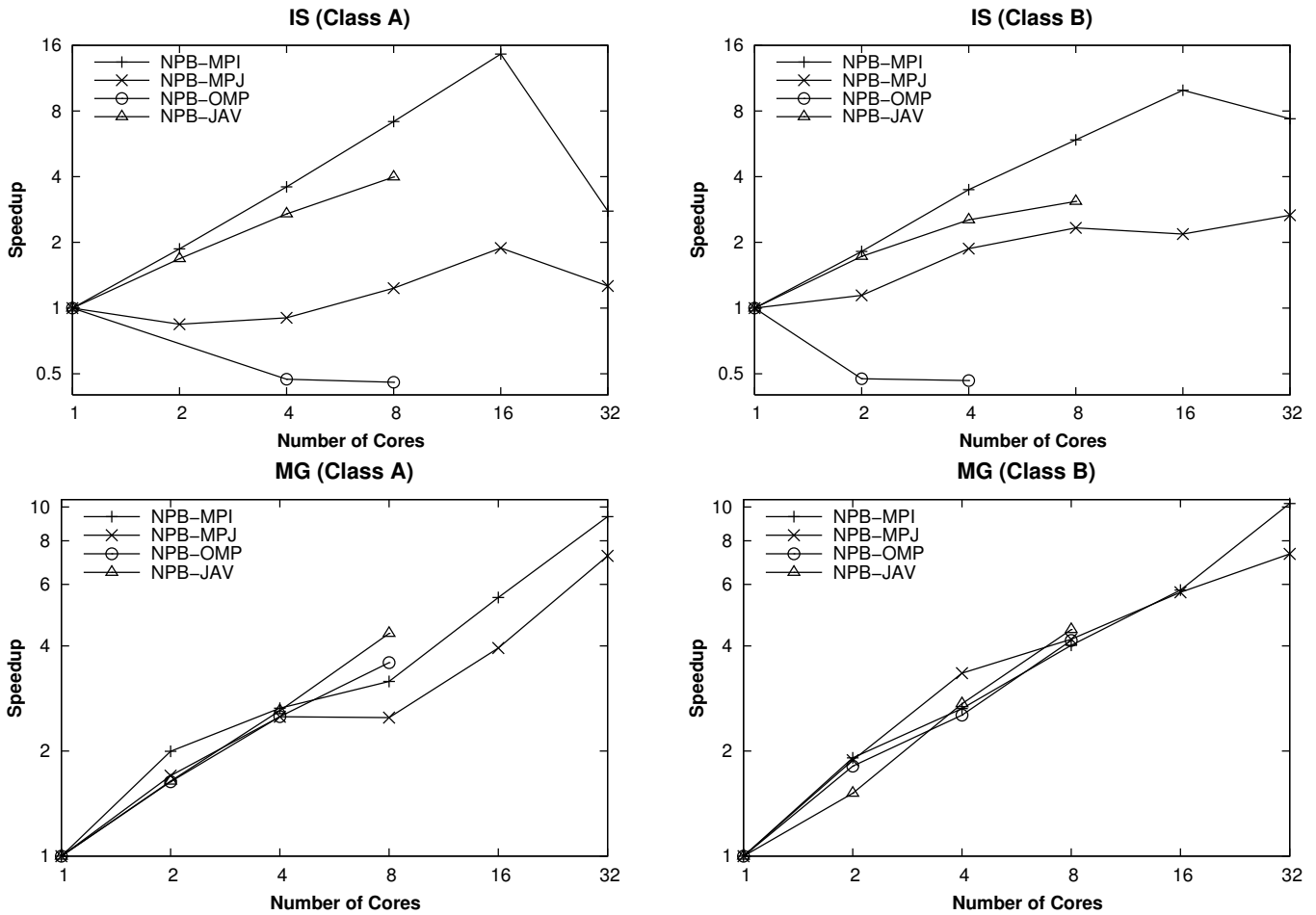
Figure 4. IS and MG kernels performance on the Infiniband multi-core cluster

B) than MPI. Regarding Figure 6 results, the MPJ speedups are poor for Class A problem size. Nevertheless, NPB-MPJ obtains much better performance for Class B workload, where it outperforms NPB-JAV and achieves up to a 50% of the NPB-MPI scalability on 25 cores.

VI. CONCLUSIONS

This paper has presented NPB-MPJ, the first extensive implementation of the standard benchmark suite, the NPB, for Message-Passing in Java (MPJ). NPB-MPJ allows, as main contributions: (1) the evaluation of the important number of existing MPJ libraries; (2) the analysis of MPJ performance against other Java parallel approaches; (3) the assessment of MPJ versus native MPI performance; and (4) the compilation of Java optimization techniques for parallel programming, from which NPB-MPJ has especially benefited, obtaining significant performance increases.

NPB-MPJ has been used in the performance evaluation of two multi-core clusters. The analysis of the results has shown that MPJ libraries are an alternative to native languages (C/Fortran) for parallel programming multi-core systems. However, significant performance bottlenecks have been detected with the aid of NPB-MPJ, which can help MPJ

libraries developers to boost MPJ performance and bridge the gap with native solutions. Furthermore, a more detailed performance analysis would include a breakdown, easily supported in NPB-MPJ, of the total time spent into computation and communication.

REFERENCES

[1] "Message Passing Interface Forum," http://www.mpi-forum.org [Last visited: November 2008].

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

[3] INRIA, "ProActive Website," http://proactive.inria.fr [Last visited: November 2008].

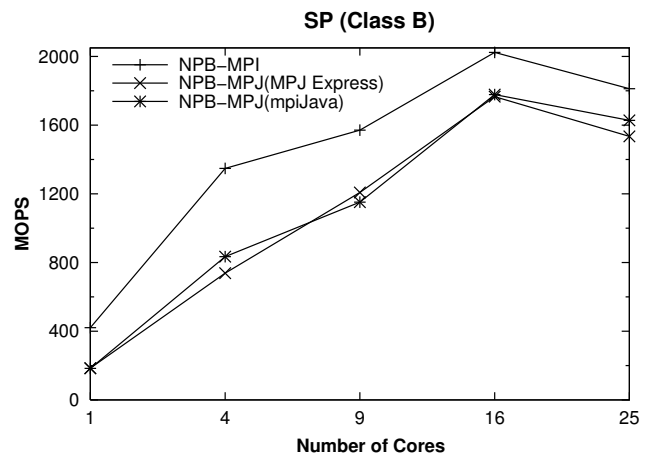[4] "NAS Parallel Benchmarks," http://www.nas.nasa.gov/Resources/Software/npb.html [Last visited: November 2008].
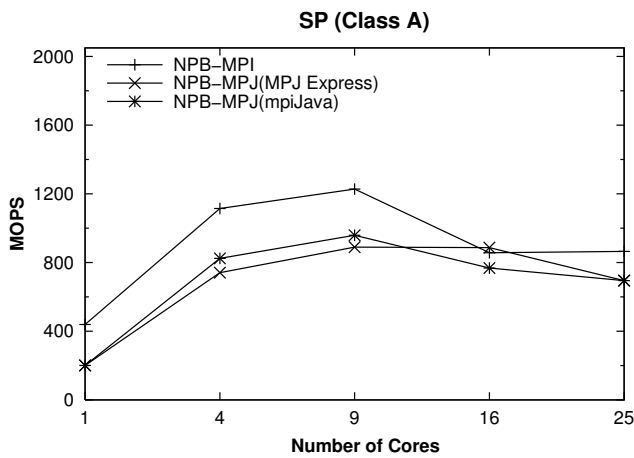
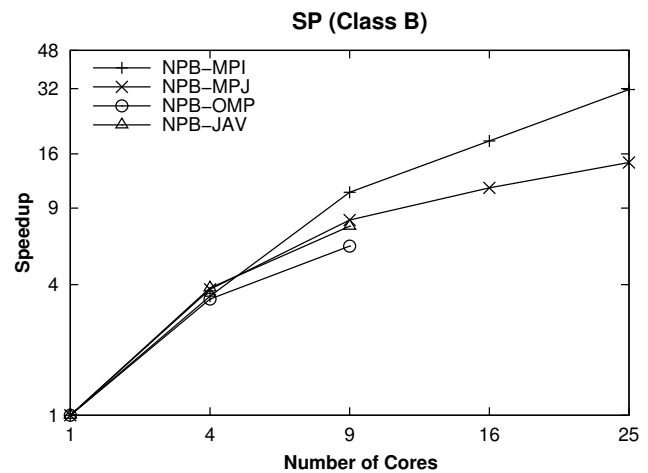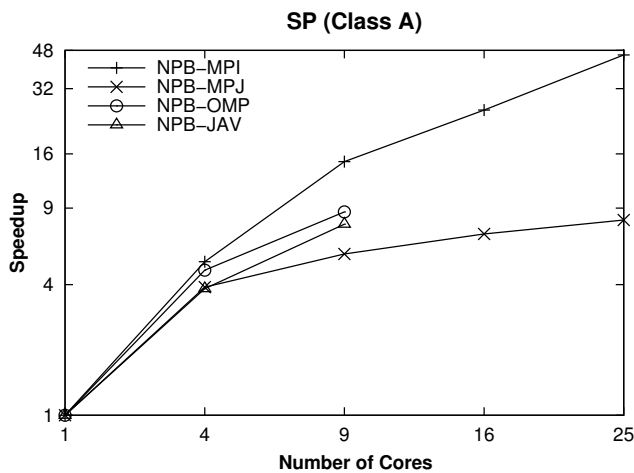Figure 5. NPB SP performance on the Gigabit Ethernet multi-core cluster

Figure 6. NPB SP performance on the Infiniband multi-core cluster

[5] 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)*, Hong Kong, China, 2003, pp. 118–126.

[6] B. Carpenter, G. Fox, S.-H. Ko, and S. Lim, "mpiJava 1.2: API Specification," http://www.hpjava.org/reports/mpiJava-spec/ [Last visited: November 2008].

[7] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox, "MPJ: MPI-like Message Passing for Java," *Concurrency: Practice and Experience*, vol. 12, no. 11, pp. 1019–1038, 2000.

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

[9] M. Baker, B. Carpenter, and A. Shafi, "MPJ Express: Towards Thread Safe Java HPC," in *Proc. 8th IEEE Intl. Conf. on Cluster Computing (CLUSTER'06), Barcelona, Spain*, 2006, pp. 1–10.

[10] M. Bornemann, R. V. van Nieuwpoort, and T. Kielmann, "MPJ/Ibis: A Flexible and Efficient Message Passing Platform for Java," in *12th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'05), Sorrento, Italy*, 2005, pp. 217–224.

[11] 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*, vol. 17, no. 7-8, pp. 1079–1107, 2005.

[12] 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 (IWJPDC'07)*, Long Beach, CA, 2007, p. 196a (8 pages).

[13] 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*, vol. 18, no. 12, pp. 1541–1557, 2006.

[14] S. Genaud and C. Rattanapoka, "A Peer-to-Peer Framework for Robust Execution of Message Passing Parallel Programs," in *12th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'05), Sorrento, Italy*, 2005, pp. 276–284.

[15] B. Amedro, D. Caromel, F. Huet, and V. Bodnartchouk, "Java ProActive vs. Fortran MPI: Looking at the Future of Parallel Java," in *Proc. 10th Intl. Workshop on Java and Components for Parallelism, Distribution and Concurrency (IWJPDC'08), Miami, FL*, 2008, p. 134b (8 pages).

[16] 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)*, Hawthorne, NY, 2005, pp. 200–214.

[17] 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*, vol. 12, no. 6, pp. 375–388, 2000.

[18] V. Getov, Q. Lu, M. Thomas, and M. Williams, "Message-passing Computing with Java: Performance Evaluation and Comparisons," in *Proc. 9th Euromicro Workshop on Parallel and Distributed Processing (PDP'01)*, Mantova, Italy, 2001, pp. 173–177.

[19] 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)*, College Station, TX, 2003, pp. 323–339.

[20] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence, "Java Programming for High-Performance Numerical Computing," *IBM Systems Journal*, vol. 39, no. 1, pp. 21–56, 2000.