High-Performance Parallel Programming in Java: Exploiting Native Libraries

Vladimir Getov[1], Susan Flynn Hummel[2], Sava Mintchev[1]

[1]School of Computer Science, University of Westminster, Harrow HA1 3TP, UK
(email: {v.s.getov, s.m.mintchev}@westminster.ac.uk)
[2]IBM T. J. Watson Research Center, Yorktown Heights, NY10598, USA
(email: hummel@ebbets.poly.edu)

Abstract

With most of today's fast scientific software written in Fortran and C, Java has a lot of catching up to do. In this paper we discuss how new Java programs can capitalize on high-performance libraries for other languages. With the help of a tool we have automatically created Java bindings for several standard libraries: MPI, BLAS, BLACS, PBLAS, ScaLAPACK. The purpose of the additional software layer introduced by the bindings is to resolve the interface problems between different programming languages such as data type mapping, pointers, multidimensional arrays, etc. For evaluation, performance results are presented for Java versions of two benchmarks from the NPB and PARKBENCH suites on the IBM SP2 using JDK and IBM's high-performance Java compiler, and on the Fujitsu AP3000 using Toba — a Java-to-C translator. The results confirm that fast parallel computing in Java is indeed possible.

1 INTRODUCTION

The fundamental trade-off between portability and performance is well known to writers of high-performance scientific applications. The Java language designers placed an emphasis on portability (and in particular, mobility) of code in favour of performance. That is why making Java programs run fast on parallel machines is not an easy task!

A common solution to the performance *vs.* portability conundrum is the standard library approach. The library interface is standardised, but multiple architecture-specific performance-tuned implementations of the same library are created on different computer platforms. In that way programs using a library can be both portable and efficient across a wide range of machines.

Standard libraries often used for high-performance scientific programming include the *Message-Passing Interface* (MPI), and the *Scalable Linear Algebra PACKage* (ScaLAPACK). MPI [16] has over 120 different operations, ranging from primitive point-to-point communication functions to powerful collective operations, as well as built-in support for writing new libraries. In addition, it is thread-safe by design, and caters for heterogeneous networks. MPI aims to ensure portability of software without introducing a substantial performance penalty, and has vendor-supplied implementations on a range of hardware architectures.

ScaLAPACK [3] is a high-performance linear algebra package built on top of a message-passing library like MPI. It has been designed to be both efficient and portable, by restricting most machine dependency to two standard libraries: the *Basic Linear Algebra Subprograms* (BLAS), and the *Basic Linear Algebra Communication Subprograms* (BLACS). ScaLAPACK is built on top of these two libraries, as well as LAPACK and Parallel BLAS (PBLAS). The package is used for solving a large class of scientific problems, including systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems.

Providing access to libraries like MPI and ScaLAPACK seems imperative if Java is to achieve the success of Fortran and C in scientific programming. Access to standard libraries is essential not only for performance reasons, but also for software engineering considerations: it would allow the wealth of existing Fortran and C code to be reused at virtually no extra cost when writing new applications in Java.

The Java language has several built-in mechanisms which allow the parallelism inherent in scientific programs to be exploited. Threads and concurrency constructs are well-suited to shared-memory computers, but not large-scale distributed-memory machines. Sockets and the *Remote Method Invocation* (RMI) interface allow network programming; but they are rather low-level to be suitable for SPMD-style scientific programming, and code based on them would potentially underperform platform-specific implementations of standard communication libraries like MPI.

In this paper we present a way of successfully tackling the difficulties of binding native libraries to Java.

2 CREATING NATIVE LIBRARY WRAPPERS

In principle, the binding of a native library to Java amounts to either dynamically linking the library to the Java virtual machine, or linking the library to the object code produced by a stand-alone Java compiler. Complications stem from the fact that Java data formats are in general different from those of C. Java implementations have a *native method interface* which allows C functions to access Java data and perform format conversion if necessary. Such an interface is fairly convenient for writing *new* C code to be called from Java, but is not adequate for linking *existing* native code.

Binding a native library to Java may also be accompanied by portability problems. The native interface was not part of the original Java language specification [11], and different vendors have offered incompatible interfaces. The *Java Native Interface* (JNI) in Sun's JDK 1.1 is now the definitive interface, but it is not yet supported in all Java implementations on different platforms by other vendors. Thus to maintain the portability of the binding one may have to cater for a variety of native interfaces.

Clearly an additional interface layer must be written in order to bind a native library to Java. A large library like MPI can have over a hundred exported functions, therefore it is preferable to automate the creation of the additional interface layer. The *Java-to-C interface generator* (JCI) [15] takes as input a header file containing the C function prototypes of the native library. It outputs a number of files comprising the additional interface:

- a file of C stub-functions;
- files of Java class and native method declarations;
- shell scripts for doing the compilation and linking.

The JCI tool generates a C stub-function and a Java native method declaration for each exported function of the native library. Every C stub-function takes arguments whose types correspond directly to those of the Java native method, and converts the arguments into the form expected by the C library function.

3 THE JAVA BINDING FOR MPI

The largest native library we have bound to Java so far is MPI — it has in excess of 120 functions [15]. The JCI tool allowed us to bind all those functions to Java without extra effort. Since the MPI specification is a *de facto* standard, the binding generated by JCI should be applicable without modification to *any* MPI implementation. As the Java binding for MPI has been generated automatically from the C prototypes of MPI functions, it is very close to the C binding. This similarity means that the Java binding is almost completely documented by the MPI-1 specification, with the addition of a table of the JCI mapping of C types into Java types.

3.1 Addresses in JavaMPI and derived MPI data types

In Java the passing of addresses to/from native functions must be handled with care because of the strictness of the type system, and the automatic memory management. Addresses in the C binding for MPI have the type `MPI_Aint` which is a synonym for `long int`; that type is used for both relative and absolute addresses. In JavaMPI the situation is different: relative addresses are of type `int`, while absolute addresses are of type `Object`. It must be stressed that the absolute addresses of objects should never be stored or passed as `int` values to native functions because the garbage collector would then be free to relocate (or even deallocate) the objects.

Thus MPI functions which expect or return an `MPI_Aint` can be used in JavaMPI with either `int` or `Object`. Table 1 lists the permitted uses of those functions. Where a function has both an absolute and relative form, it is up to the programmer to select the appropriate alternative.

Table 1: Functions with MPI_Aint arguments in Java

Absolute address	**Relative address**	**Either**
`MPI_Aint` → `Object`	`MPI_Aint` → `int`	
MPI_Address	MPI_Type_extent	MPI_Type_lb
	MPI_Type_size	MPI_Type_ub
		MPI_Type_hvector
		MPI_Type_hindexed
		MPI_Type_struct

Formal arguments used as data buffers in JavaMPI are of type `Object`, corresponding to `(void *)` in the C binding. Unlike most other communication libraries, MPI provides a mechanism (*derived data types*) which allows non-contiguous data items (possibly of different types) to be included in a single message, without moving the data into a contiguous buffer. The operations in Table 1 are used for constructing derived data types. MPI data structures and other derived types must be used in conformance with the Java data layout as described in the language specification [11]. Example 3.1 shows how two separate Java objects can be bundled together in one message by means of an MPI structure type.

Example 3.1 *Broadcasting an MPI data structure*

```
ObjectOfJint  rewsna  = new ObjectOfJint(0),
              answer  = new ObjectOfJint(0);

MPI_Datatype struct   = new MPI_Datatype ();
int   block_lengths[] = {1, 1};
Object  displs []     = {answer, rewsna};
MPI_Datatype types [] = {constMPI.MPI_INT, constMPI.MPI_INT};

javaMPI.MPI_Type_struct (2, block_lengths, displs, types,
                         JCI.ptr(struct));
javaMPI.MPI_Type_commit (JCI.ptr(struct));
javaMPI.MPI_Bcast       (constMPI.MPI_BOTTOM, 1, struct, 0,
                         constMPI.MPI_COMM_WORLD);
javaMPI.MPI_Type_free   (JCI.ptr(struct));
```

The addresses of the two objects, answer and rewsna are stored into the array of displacements displs. The MPI_Type_struct call creates an MPI data structure containing the two objects, which is then broadcast to all processes. JCI.ptr is an overloaded method generated by JCI, and is an analog of the & operator in C. The MPI_Type_commit and MPI_Type_free calls instruct MPI respectively to build and release the derived data type.

3.2 Array arguments to native MPI functions

In both C and Fortran-77, one can pass the address of an array element as an actual argument to a function or subroutine. This is not possible in Java. If one needs to use an array starting at a certain offset as an MPI data buffer, then an MPI data type can help. Example 3.2 shows how to broadcast 20 elements of an integer array (block) starting with the 10^{th} element.

Example 3.2 *Describing an array section by an MPI data type*

```
MPI_Datatype struct = new MPI_Datatype ();
int   block_lengths[] = {20};
long  displs []       = {10};
MPI_Datatype types [] = {constMPI.MPI_Int};

javaMPI.MPI_Type_struct (1,block_lengths,displs,types,JCI.ptr(struct));
javaMPI.MPI_Type_commit (JCI.ptr(struct));
javaMPI.MPI_Bcast (block, 1, struct, 0, constMPI.MPI_COMM_WORLD);
javaMPI.MPI_Type_free   (JCI.ptr(struct));
```

Using array buffers with JavaMPI requires an understanding of Java data layout. In particular, an array of arrays in Java is in fact an array of pointers to array objects, and not one contiguous two-dimensional array. Hence an array of arrays should be described by MPI_Type_hindexed, and not by MPI_Type_contiguous as the case would be in C.

Example 3.3 *Using a two-dimensional array as a data buffer*

```
int           arr[][] = {{0, 1}, {2, 3}};
MPI_Datatype arr_type = new MPI_Datatype ();
int block_lengths []  = {2, 2};

javaMPI.MPI_Type_hindexed (2, block_lengths, arr,
                           constMPI.MPI_INT, JCI.ptr(arr_type));
javaMPI.MPI_Type_commit   (JCI.ptr(arr_type));
javaMPI.MPI_Bcast (constMPI.MPI_BOTTOM, 1, arr_type, 0,
                   constMPI.MPI_COMM_WORLD);
javaMPI.MPI_Type_free     (JCI.ptr(arr_type));
```

Notice that in Example 3.3 the two-dimensional array buffer arr is used as its own indexing array in the call to javaMPI.MPI_Type_hindexed.

4 BINDINGS FOR SCALAPACK AND ITS CONSTITUENT LIBRARIES

ScaLAPACK is built on top of a number of libraries written in either Fortran-77 or C. The C libraries for which we have created Java bindings are the *Parallel Basic Linear Algebra Subprograms* (PBLAS) and the *Communication Subprograms* (BLACS). The library function prototypes have been taken from the PARKBENCH 2.1.1 distribution [17].

Table 2: Native libraries bound to Java

library	written in	Size of Java binding functions	C lines	Java lines
MPI	C	125	4434	439
BLACS	C	76	5702	489
BLAS	F77	21	2095	169
PBLAS	C	22	2567	127
PB-BLAS	F77	30	4973	241
LAPACK	F77	14	765	65
ScaLAPACK	F77	38	5373	293

The JCI tool can be used to generate Java bindings for libraries written in languages other than C, provided that the library can be linked to C programs, and prototypes for the library functions are given in C. We have created Java bindings for the ScaLAPACK constituent libraries written in Fortran-77: BLAS Level 1–3, PB-BLAS, LAPACK, and ScaLAPACK itself. The C prototypes for the library functions have been inferred by *f2c* [10].

Table 2 gives some idea of the sizes of JCI-generated bindings for individual libraries. In addition, there are some 2280 lines of Java class declarations produced by JCI which are common to all libraries. The automatically generated bindings are fairly large in size because they are meant to be portable, and to support different data formats. On a particular hardware platform and Java native interface, much of the binding code may be eliminated during the preprocessing phase of its compilation.

Even though JCI does a lot to smooth out the interface between Java and native libraries, calling native library functions may not be as straightforward and elegant as calling Java functions. Some peculiarities and difficulties encountered while writing Java programs which access native libraries are listed below.

Pointers/addresses. A pointer to a value of type *j_type* is represented in JCI-generated bindings as a class with a single field `val` of type *j_type*. Pointer objects can be created and initialized by Java constructors, or by the overloaded function `JCI.ptr`; they can be dereferenced by accessing the `val` field.

There is some peculiarity when accessing a Fortran native library. Arguments in Fortran-77 are always passed by address, therefore all scalar arguments to a Fortran native function must be enclosed in pointer objects, regardless of whether they are intended for input or output of values.

Array offsets. In both C and Fortran-77, one can pass the address of an array element as an actual argument to a function or subroutine. This is not possible in Java; therefore a Java program cannot pass part of an array starting at a certain offset to a (native library) function. One way round this restriction, applied in the Java Linpack benchmark [9, 2], is to add one integer "offset" argument for each array argument of a function. The JCI-generated bindings support a more elegant solution as well, which does not involve extra arguments to native library functions. The elements of an array *arr* of any type starting at offset i can be passed to a native library function by

`JCI.section` (*arr*, i)

where `JCI.section` is an overloaded method whose definition is generated by JCI.

Example 4.1 *Passing an array section to a native library function*

```
blas.idamax (JCI.ptr(n-k), JCI.section(col_k, k), one)
```

The array `col_k` starting at offset `k` is passed to the BLAS function `idamax`. Type safety with `JCI.section` is guaranteed: the compiler will check if the array has the required type. Example 4.1 also illustrates one unfortunate consequence of accessing a Fortran function discussed above: all scalars must be passed by address (*i.e.* be wrapped in objects, for example by `JCI.ptr`).

Multidimensional arrays. Many native library functions take multidimensional arrays (*e.g.* matrices) as arguments. The JCI tool supports multidimensional arrays, but a run-time penalty is incurred: such arrays must always be relocated in memory in order to be made contiguous before being supplied to a native function. When large data arrays are involved the inefficiency can be significant. In order to avoid it, we have chosen to represent matrices in Java as one-dimensional arrays, for all native libraries of Section 4. In JavaMPI (Section 3) multi-dimensional arrays are left intact without significant inefficiency. Large arrays used as data buffers can have their layout described by an MPI derived data type (Example 3.3), and the Java binding performs no conversion for them. Multi-dimensional arrays used as descriptors are normally relatively small.

Another point to bear in mind when accessing a Fortran-77 library is the inverse order of array layout in comparison with C.

Array indexing. This problem is specific to native libraries written in Fortran, where arrays are normally indexed starting from 1, while in Java as in C indices start from 0. Java programs calling Fortran native functions that receive or return an array index must be aware of the difference. An example of such a function is IDAMAX from BLAS (Example 4.1).

5 EXPERIMENTAL RESULTS

In order to evaluate the performance of the Java binding to native libraries, we have translated into Java a benchmark written in C and MPI — the Integer Sort (IS) kernel from the NAS Parallel Benchmark suite NPB2.2 [1]. The program sorts in parallel an array of N integers. The problem size (N) is $N = 8$M for IS Class A. The original C and the new Java versions of IS are quite similar, which allows a meaningful comparison of performance results.

We have run the IS benchmark on different platforms: a cluster of Sun Sparc workstations, a Fujitsu AP3000 (UltraSPARC 167 MHz nodes) at the Fujitsu European Center for Information Technology (FECIT), and the IBM SP2 system at the Cornell Theory Center (120 MHz P2SC processors). Some performance results obtained on the SP2 machine are shown in Table 3 while the benchmark measurements on the AP3000 are presented in Table 4. The Java implementations used are IBM's port of JDK 1.0.2D [14] (with the JIT compiler enabled), IBM's Java compiler *hpcj* [13] (with flags *-O* and *jnoruncheck*), and Toba 1.0.b6 [18] of The University of Arizona. The *hpcj* optimizing compiler generates native RS/6000 code, while Toba translates Java byte code into C. The MPI libraries are LAM 6.1 and Fujitsu MPIAP. We opted for LAM rather than the proprietary IBM MPI library because the version of the latter available to us (PSSP 2.1) does not support the re-entrant C library required for Java. A fully thread-safe implementation of MPI has not been required in our experiments so far since MPI is used in a single-threaded way.

To identify the causes of the slowdown of the Java (JDK) version of IS with respect to the C version we have instrumented the JavaMPI binding, and gathered additional measurements. It turns out that the cumulative time each process spends in the C functions of the JavaMPI binding is approximately 20 milliseconds regardless of the number of processors, and thus has a negligible share in the breakdown of the total execution time for the Java version of IS. Clearly the JavaMPI binding does not introduce a noticeable overhead in the results from Table 3 and Table 4.

Table 3: Execution statistics for the C and Java versions of the IS benchmark (Class A) on IBM SP2

Lang	MPI implementation	Execution time (s) 1	2	4	8	16	Performance (Mflop/s) 1	2	4	8	16
		No of processors					No of processors				
JDK	LAM	—	48.04	24.72	12.78	6.94	—	1.75	3.39	6.56	12.08
hpcj	LAM	—	23.27	13.47	6.65	3.49	—	3.60	6.23	12.62	24.01
C	LAM	42.16	24.52	12.66	6.13	3.28	1.99	3.42	6.63	13.69	25.54
C	IBM	40.94	21.62	10.27	4.92	2.76	2.05	3.88	8.16	14.21	30.35

Table 4: Execution statistics for the C and Java versions of the IS benchmark (Class A) on Fujitsu AP3000

Lang	MPI implementation	Execution time (s) 1	2	4	8	Performance (Mflop/s) 1	2	4	8
		No of processors				No of processors			
Toba	LAM	79.58	49.75	27.03	13.44	1.05	1.69	3.10	6.24
Toba	MPIAP	86.79	48.71	24.69	12.42	0.97	1.72	3.40	6.75
C	LAM	42.55	27.38	15.21	9.04	1.97	3.06	5.52	9.28
C	MPIAP	49.87	26.57	14.15	7.65	1.68	3.16	5.93	10.97

The performance of Java IS programs compiled with *hpcj* is very impressive, and provides evidence that the Java language can be used successfully in high-performance computing.

Further experiments have been carried out with a Java translation of the Matrix Multiplication (MATMUL) benchmark from the PARKBENCH suite [17]. The original benchmark is in Fortran-77 and performs dense matrix multiplication in parallel. It accesses the BLAS, BLACS and LAPACK libraries included in the PARKBENCH 2.1.1 distribution. MPI is used indirectly through the BLACS native library. The default problem size (N) is $N = 1000$. We have run MATMUL on a Sparc workstation cluster, and on the IBM SP2 machine at Southampton University (66MHz Power2 "thin1" nodes). The results are shown in Table 5.

Table 5: Execution statistics for the Fortran and Java versions of the MATMUL benchmark ($N = 1000$) on the IBM SP2

Lang	MPI implementation	Execution time (s) 1	2	4	8	16	Performance (Mflop/s) 1	2	4	8	16
		No of processors					No of processors				
JDK	LAM	—	17.09	9.12	5.26	3.53	—	117.0	219.4	380.2	566.9
F77	LAM	—	16.45	8.61	5.12	3.13	—	121.6	232.3	390.4	638.3
F77	IBM	33.25	15.16	7.89	3.91	2.20	60.16	132.0	253.6	511.2	910.0

It is evident from Table 5 that Java MATMUL execution times are only 5–10% longer than Fortran-77 times. These results may seem surprisingly good, given that Java IS under JDK is about two times slower than C IS (Table 3). The explanation is that in MATMUL most of the performance-sensitive calculations are carried out by the native library routines rather than by the program. In contrast, IS uses a native library (MPI) only for communication, and all computations are done by the benchmark program.

6 DISCUSSION AND RELATED WORK

Until the Java compiler technology reaches maturity, the use of native numerical code in Java programs is certain to improve performance, as recent experiments with the Java Linpack benchmark [9] and several BLAS Level 1 functions written in C have shown [2]. By binding the original native libraries like BLAS, Java programs can gain in performance on all those hardware platforms where the libraries are efficiently implemented.

A Java-to-PVM interface is publicly available [19]. A Java binding for a subset of MPI has also been written [4, 7] and run on up to 8 processors of Sun Ultra Sparc. In comparison, our binding has been generated automatically to cover the whole of MPI; it allows for better use of MPI derived data types; its flexibility makes it easier to retarget to different versions of the Java native interface and hardware platforms. A complete draft definition of a Java Binding for MPI [5] along the lines of the MPI-1.2 C++ binding has been proposed, and work on an implementation is under way. A commercial effort to develop an MPI framework and parallel support environment for Java has been started recently by MPI Software Technology, Inc. [8].

The approach of binding native libraries to Java has certain limitations. In particular, for security reasons applets downloaded over the network may not load libraries or define native methods. Furthermore, a Java program can only use a native library in a single threaded manner unless that library is thread-safe.

Some closely related work involves the automatic translation of existing Fortran library code to Java with the help of a tool like *f2j* [6]. This approach offers a very important long-term perspective as it preserves Java portability, while achieving high performance in this case would obviously be more difficult.

A different way of employing Java in high performance computing is to utilize the potential of Java concurrent threads for programming parallel shared-memory machines. A very interesting related theme is the implementation on the IBM POWERparallel System SP machine of a Java run-time system with parallel threads [12], using message passing to emulate shared memory. The run-time system would eventually be written in Java with MPI.

MPI offers message-passing operations portable over a large variety of machines, but has not to date been bound to many languages. The MPI-1 standard [16] includes bindings for just two languages — C and Fortran-77. C++ bindings are included in the MPI-2 document for both MPI-1 and MPI-2.

7 CONCLUSIONS

In this paper we have summarised our work so far on enabling high performance computations in Java. We have used a prototype tool for automating the creation of portable interfaces to native libraries (whether for scientific computation or message passing). We have applied the JCI tool to create Java bindings for MPI, BLAS, ScaLAPACK etc. which are fully compatible with the library specifications. The bindings are publicly available from `http://perun.hscs.wmin.ac.uk/CSPE/software.html`. With performance-tuned implementations of those libraries available on different machines, and with compilers like IBM's *hpcj* that generate fast native code, efficient numerical programming in Java is now possible.

ACKNOWLEDGEMENTS

The experiments have been carried with the support of the Cornell Theory Center, the University of Southampton, and the Fujitsu European Center for Information Technology.

We are grateful to Tony Hey and David Snelling for their help and comments. Special thanks are also due to the anonymous referees for their valuable suggestions.

References

[1] D. Bailey et al. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, 1994. http://science.nas.nasa.gov/Software/NPB.

[2] A.J.C. Bik and D.B. Gannon. A note on native Level 1 BLAS in Java. *Concurrency: Practice and Experience*, 9(11):1091–1099, 1997.

[3] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. ScaLAPACK: A linear algebra library for message-passing computers. In *SIAM Conference on Parallel Processing*, 1997.

[4] B. Carpenter, Y-J. Chang, G. Fox, D. Leskiw, and X. Li. Experiments with "HPJava". *Concurrency: Practice and Experience*, 9(6):633–648, 1997.

[5] B. Carpenter, G. Fox, G. Zhang, and X. Li. A draft Java binding for MPI. http://www.npac.syr.edu/projects/pcrc/doc/, November, 1997.

[6] H. Casanova, J.J. Dongarra, and D.M. Doolin. Java access to numerical libraries. *Concurrency: Practice and Experience*, 9(11):1279–1292, 1997.

[7] Y-J. Chang and B. Carpenter. MPI Java wrapper download page. http://www.npac.syr.edu/users/yjchang/javaMPI. March, 1997.

[8] G. Crawford III, Y. Dandass, and A. Skjellum. The JMPI commercial message passing environment and specification: Requirements, design, motivations, strategies, and target users. http://www.mpi-softtech.com/publications/JMPI_121797.html, MSTI, December 1997.

[9] J. Dongarra and R. Wade. Linpack benchmark – Java version. http://www.netlib.org/benchmark/linpackjava, 1996.

[10] S. I. Feldman and P. J. Weinberger. *A Portable Fortran 77 Compiler. UNIX Time Sharing System Programmer's Manual, Tenth Edition.* AT&T Bell Laboratories, 1990.

[11] J. Gosling, W. Joy, and G. Steele. *The Java Language Specification, Version 1.0.* Addison-Wesley, Reading, Mass., 1996.

[12] S.F. Hummel, T. Ngo, and H. Srinivasan. SPMD programming in Java. *Concurrency: Practice and Experience*, 9(6):621–631, 1997.

[13] IBM Corp. *High-performance compiler for Java: An optimizing native code compiler for Java applications.* http://www.alphaWorks.ibm.com/formula, 1997.

[14] IBM Corp. *JDK for AIX, IBM Centre for Java Technology Development.* http://ncc.hursley.ibm.com/javainfo/hurindex.html, 1997.

[15] S. Mintchev and V. Getov. Towards portable message passing in Java: Binding MPI. In *Proceedings of EuroPVM-MPI*, pages 135–142, Kraków, Poland, November, 1997. Springer LNCS 1332.

[16] MPI Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[17] PARKBENCH Committe (assembled by R. Hockney and M. Berry). PARKBENCH report - 1: Public international benchmarks for parallel computers. *Scientific Programming*, 3(2):101–146, 1994. http://www.netlib.org/parkbench.

[18] T.A. Proebsting, G. Townsend, P. Bridges, J.H. Hartman, T. Newsham, and S.A. Watterson. Toba: Java for applications — a way ahead of time (WAT) compiler. In *Proceedings Third Conference on Object-Oriented Technologies and Systems (COOTS'97)*, Portland, Oregon, 1997.

[19] D.A. Thurman. JavaPVM: The Java to PVM interface. http://www.isye.gatech.edu/chmsr/JavaPVM, 1996.