A MIXED-LANGUAGE PROGRAMMING METHODOLOGY FOR HIGH PERFORMANCE JAVA COMPUTING*

Vladimir S. Getov
University of Westminster
Northwick Park, Harrow, UK
and
Los Alamos National Laboratory
Los Alamos, NM, USA

Abstract Java is quickly becoming the most popular platform for distributed computing. However, its performance is still subject to concerns in comparison to other programming languages such as C and Fortran. As a consequence, programmers of high-performance applications are usually reluctant to embrace Java as an alternative language in their work. This article introduces the Java-to-C Interface (JCI) tool which generates automatically the wrapper code interfacing existing scientific libraries to Java. Thus, facilitating rapid development and software reuse, the JCI tool provides application programmers with immediate accessibility to existing scientific libraries from Java. While beneficial to the software developer, the additional advantages of mixed-language programming in terms of application performance are addressed in detail within the context of this work. We also present analysis and comparisons of evaluation results for mixed-language codes in Java and C/Fortran on a high-performance distributed memory computer (IBM SP-2). The NAS Embarrassingly Parallel and Integer Sort benchmarks as well as the Matrix Multiplication kernel from the PARKBENCH suite were selected for our experiments. The evaluation results demonstrate the feasibility and efficiency of our mixed-language programming methodology with Java.

Keywords: Java, mixed-language programming, high-performance computing, automatic wrapper generation.

*Research supported in part by Higher Education Funding Council for England (UK) under the NFF program and by Los Alamos National Laboratory (USA) under the Exchange Visitors program.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35407-1_22

R. F. Boisvert et al. (eds.), *The Architecture of Scientific Software*

1. INTRODUCTION

One of the problems facing the Java programming language and its acceptance for scientific computing is performance. In general, the fundamental trade-off between portability and performance is very well known to the high-performance computing community. The Java language designers placed an emphasis on portability (and in particular, mobility) of code in favour of performance. This is one of the main reasons why making Java programs run fast is not an easy task.

A closer inspection shows that the Java platform 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. Although sockets and the Remote Method Invocation (RMI) interface allow network programming, they are rather low-level to be suitable for the Single-Program-Multiple-Data (SPMD) parallel programming model. Therefore, codes based on them would potentially underperform platform-specific high-performance implementations of standard scientific and communication libraries.

Nevertheless, as a programming language, Java has the core qualities needed for writing high-performance applications. With the maturing of compilation technology, such applications written in Java are starting to appear. Fortunately, rapid progress is being made in this area by developing static Java compilers, such as the IBM High-Performance Compiler for Java (HPCJ), which generates optimized native code for the RS6000 architecture [11]. Since the Java language is relatively new, however, it lacks the extensive scientific libraries of other languages such as Fortran and C. This is one of the major obstacles towards efficient and user-friendly computationally intensive programming in Java.

Standard libraries often used for high-performance scientific computing include the Message Passing Interface (MPI), and the Scalable Linear Algebra PACKage (ScaLAPACK). Providing access to such libraries 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. If available, 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. In order to overcome these problems, we have applied our JCI code generating tool to create Java bindings for various legacy libraries [7].

In this article we first describe the design principles of the JCI tool. We also introduce our methodology for mixed-language software devel-

opment with Java and demonstrate the viability of our approach on a number of performance evaluation experiments.

2. THE JCI TOOL

At first sight it appears that the binding of a native library to Java should not be a problem, as Java implementations support the Java Native Interface (JNI) via which C functions or Fortran subroutines can be called [9]. There are some hidden problems, however. Complications stem from the fact that Java data formats are in general different from those of other languages like C, C++, Fortran, etc. This obviously requires data conversion of both arguments and results in mixed-language applications. Such conversion is a natural part of the native code if both parts of a mixed-language piece of software are to be written from scratch. For legacy codes, however, an additional interface layer called *binding* or *wrapper* must be created which performs data conversion and other auxiliary functions if necessary.

In principle, the binding of a native library to Java amounts to either dynamically linking the library to the Java Virtual Machine (JVM), or linking the library to the object code produced by a static Java compiler. Binding a legacy library to Java may also be accompanied by portability problems as the JNI specification is still not fully supported in different Java implementations. Thus, in order to maintain the portability of the binding one may have to cater for a variety of native interfaces. A large legacy library like MPI, for example, can have over a hundred exported functions. Therefore, the JCI tool which generates automatically the additional interface layer plays central role in our mixed-language programming methodology. In order to call a C function from Java, the JCI tool has to supply for each formal argument of the C function a corresponding actual argument in Java. Unfortunately, the disparity between data layout in the two languages is large enough to rule out a direct mapping in general. For instance, one has to take into account that:

- primitive types in C may be of varying sizes, different from the standard Java sizes;
- there is no direct analog to C pointers in Java;
- multi-dimensional arrays in C have no direct counterpart in Java;
- C structures can be emulated by Java objects, but the layout of fields of an object may be different from the layout of a C structure;

Table 1 Mapping of compound C types into Java types

C type	Java type	Comment
`char *`	`ObjectOfChar`	- if at top level and not the type of a function;
	`String`	- otherwise.
`struct` *name* `*`	`class` *name*	
`void *`	`Object`	
c_type `*`	`ObjectOf`*j_type*	
`char []`	`String`	
c_type `[]`	*j_type* `[]`	
`struct` *name*	`class` *name*	

- C functions passed as arguments have no direct counterpart in Java.

Therefore, one has to define a specific mapping which is then implemented by the JCI tool. Table 1 shows the scheme currently used to map C types onto Java types. Primitive types are not listed in this table because they are to be found in the documentation of each JVM's native interface. C pointers are represented in a type-safe way by a family of Java classes generated by JCI. Each such class is named `ObjectOf`*j_type*, and contains a 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. In general, the defined mapping is not unique – on the contrary – there is a number of different mappings to choose from. The selection of an appropriate mapping represents an important trade-off between the extent of the performance overhead introduced by the binding on the one hand, and the ease of use of the programming interface from Java on the other.

A block diagram of JCI is shown in Figure 1. The tool 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, and shell scripts for compiling and linking the binding. 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.

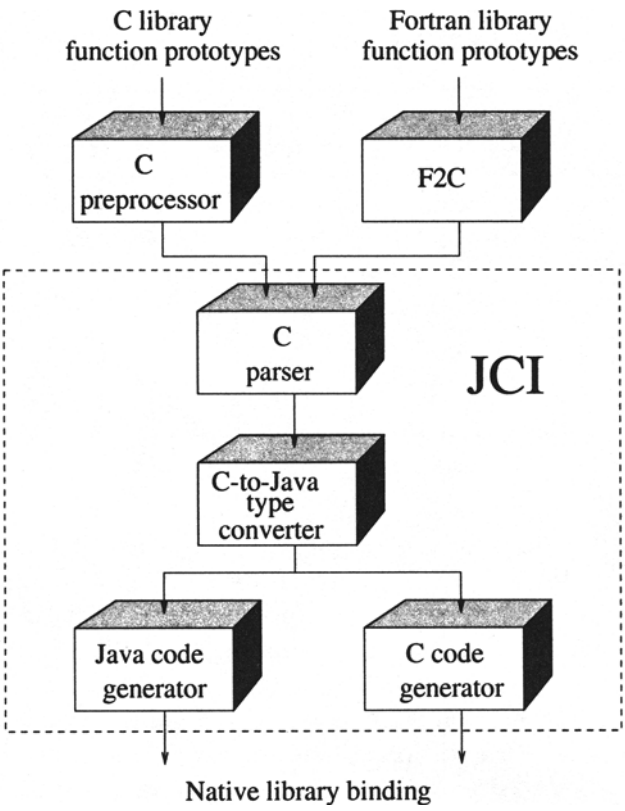

Figure 1 JCI block diagram

Thanks to the JCI tool our bindings are easily adaptable to various platforms. As we mentioned already, different Java native interfaces exist, and thus separate code may have to be generated for binding a given legacy library to different Java implementations. We have tried to limit the dependence of JCI's output on the native interface version to a set of macro definitions describing the particular native interface. Thus, it may be possible to re-bind a library to a new Java platform simply by providing the appropriate macros.

The tool also provides a good deal of flexibility for generating Java wrappers to native libraries. For example, by using different library header files as input, one can create bindings for multiple versions of a library, such as MPI-1.1, MPI-1.2, MPI-2.0. Furthermore, JCI 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. This is how we have

Table 2 Legacy 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

created Java bindings for the ScaLAPACK constituent libraries written in Fortran-77: BLAS Level 1–3, PB-BLAS, LAPACK, and ScaLAPACK itself [7]. The C prototypes for the Fortran library functions have been inferred following the methodology adopted in the Fortran–to–C translator [5].

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 cases. 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 JNI implementation, 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 legacy codes, 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. In addition, there is some specific peculiarity when accessing a Fortran native library because arguments in Fortran are always passed by reference. 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, one can pass the address of an array element as an actual argument to a function or subroutine. This is not possible in Java. Subsequently, a Java program cannot pass part of an array starting at a certain offset to a (native library) function. One way round this restriction is to add one integer "offset" argument for each array argument of a function [2]. The JCI-generated wrapper code supports 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. For example, passing an array section to a native library function can be done by

```
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, because the compiler will check if the array has the required type. This example also illustrates one unfortunate consequence of accessing a Fortran function as discussed above – all scalars must be passed by reference (*i.e.* be wrapped in objects, for example by `JCI.ptr`).

Multi-dimensional arrays. Many scientific library functions take as arguments multi-dimensional arrays such as matrices. The JCI tool supports multi-dimensional arrays, but a run-time overhead is incurred because 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 to some extent this problem, in our ScaLAPACK library bindings we have chosen to represent matrices in Java as one-dimensional arrays. On the other hand, in the Java binding for MPI [12] 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, and the Java binding performs no conversion for them. Other multi-dimensional arrays used in MPI as descriptors are relatively small and therefore not important from performance point of view.

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

3. EVALUATION RESULTS

In this section we present performance analysis and comparisons of evaluation results for both Java and C/Fortran on a high-performance distributed memory computer (IBM SP-2). The NAS parallel Embarrassingly Parallel (EP) and the Integer Sort (IS) benchmarks were used initially in our performance experiments. The EP kernel provides an estimate of the upper achievable limits for floating point performance, but requires minimal communications. The IS routine evaluates integer operations and bi-directional communications when the sorted keys are exchanged between nodes. The NAS version of IS is written in C, while the EP code is in Fortran. The NAS parallel benchmarks methodology specifies several problem sizes called "classes" in order to ensure comparative measurements across different platforms and environments. In our study, we present evaluation results for class B (2^{30} data points) of the EP kernel and class A (2^{23} data points) for the IS benchmark.

The JVM and the Java compiler used on the IBM SP-2 machine were part of the JDK for AIX. The execution environment consisted of IBM's Parallel Operating Environment (POE), which supports the loading and execution of parallel processes across the nodes of the IBM SP-2. The machine is built of thin nodes with Power-2 Super Chip (P2SC) processors and 256 Mbytes of memory on each node. The communication subsystem of the SP-2 features a high-performance switch which was used throughout the experiments. The message-passing library we have used with Java is the Local Area Multiprocessor (LAM) implementation of MPI from the Ohio Supercomputer Center [3]. Performance measurements for the corresponding Fortran or C code under both LAM and IBM's native MPI implementation are also given for comparison.

The evaluation results for the EP kernel (Figure 2) show good scalability for up to 128 nodes on the SP-2. The substantial difference is, however, the fact that benchmarks using LAM MPI for message-passing run approximately 2.5 times slower in Java than their corresponding Fortran counter part. This is not a surprise, but shows the performance penalty that one should expect from a direct port of computationally

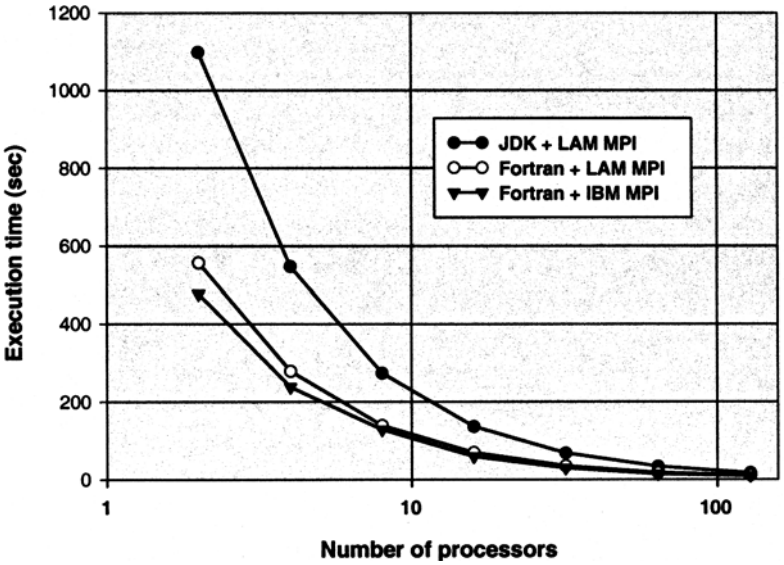

Figure 2 Execution times for the NPB EP kernel (class B) on the IBM SP-2

intensive code to Java. Of course, this is not the best mixed-language performance one can obtain as demonstrated by our further experiments.

After the initial period when the first versions of the Java platform were built for portability, the Java compiler technology has now entered a second phase where the new versions are also targeting higher performance. For example, Just-in-time (JIT) compilers have dramatically improved their efficiency, and are now challenging mature C++ compilers. Furthermore, to gain even faster execution times, the developers of HPCJ have adopted the static compilation approach [11]. Their compiler which generates native code for the RS6000 architecture was also used in this evaluation in order to compare the conventional native execution with the interpreted execution provided by JVMs.

Performance evaluation experiments with both the original C and the Java versions of the IS kernel were carried out on the IBM SP-2 machine. The results obtained are shown in Figure 3. When using the JVM for AIX for interpreted execution with the JIT compiler enabled, the Java IS benchmark is around two times slower than the original C code. In order to gain a more detailed insight and to ensure fair comparisons, we have run the C code with both the native IBM and LAM MPI implementations. As expected, the LAM-based experiment is slower but provides a basis for comparison with the Java version of the IS kernel which also uses LAM for message-passing. The performance of this

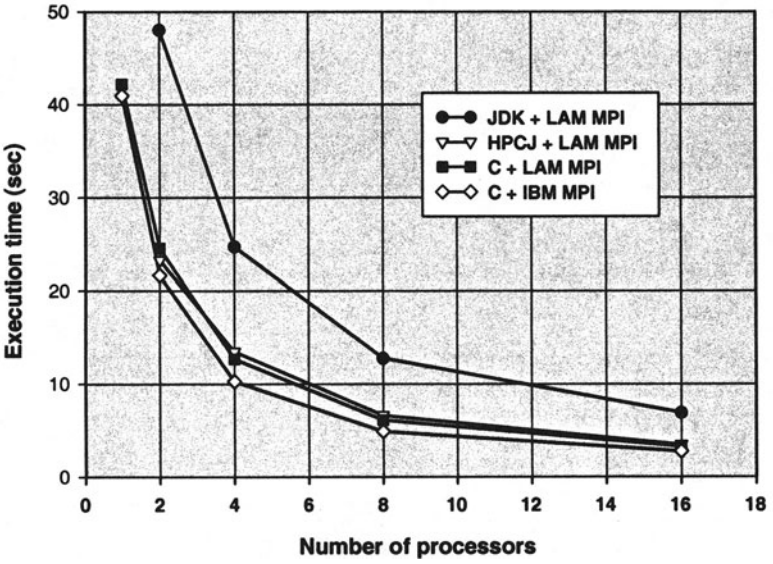

Figure 3 Execution time for the NPB IS kernel (class A) on the IBM SP-2

latter code is very impressive when compiled staticly with HPCJ. The timing results almost overlap with those delivered by the C version and provide evidence that Java can be used successfully in high-performance computing.

Further experiments on the IBM SP-2 were conducted with a Java translation of the Matrix Multiplication (MATMUL) benchmark from the PARKBENCH suite [13]. The original benchmark is written 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 also used but indirectly through the BLACS native library. The default problem size (N) is $N = 1000$.

Changing the balance between the two parts of a given code written in both Java and C or Fortran changes also the performance penalty for using Java. For example, within the MATMUL benchmark most of the performance-sensitive calculations are carried out by the native library routines rather than by the Java part of the program. Therefore, the Java MATMUL execution times are only 5–10% longer than the measurement results obtained with the original Fortran code as shown in Figure 4. In both experiments for the above comparison we have used LAM as a message-passing environment. Results obtained with the original kernel and the native IBM MPI are also given for completeness.

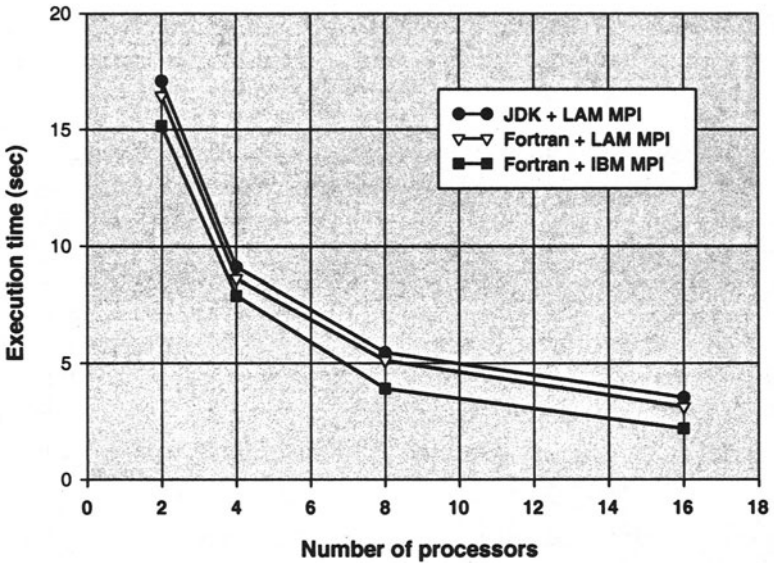

Figure 4 Execution time for the PARKBENCH MATMUL kernel on the IBM SP-2

The observations of the above experiment clearly demonstrate another dimension of flexibility for our mixed-language programming methodology. In this case, excellent performance results can be achieved even without using a static Java compiler like HPCJ. Instead, the relatively small (5–10%) performance penalty is incurred by the interpreted execution using standard JVM with the JIT compiler enabled. Such small overhead can be achieved by keeping the calculation-intensive code within the native library.

Thus, one can apply the JCI tool to wrap up the time-consuming part of a software system as a native library and implement the rest of it in Java. In such cases, the inefficient interpreted execution of the JVM is only used for a front-end Java code that provides coordination functions and interactive interfaces. Clearly, our mixed-language programming methodology does not impose any restrictions or requirements regarding the implementation level of the wrapper code. This gives the flexibility to select the most appropriate and efficient balance of different programming languages within each individual software development project.

4. DISCUSSION AND RELATED WORK

Many research groups and vendors are pursuing research to improve Java's performance which would enable more scientific and engineering

applications to be solved on Java platforms. The need for access to legacy libraries is one of the burning problems in this area. Several approaches can be taken in order to make the libraries available from Java:

- Rewriting by hand existing libraries in Java. Considering the size of the available codes and the number of years that were invested in their development, rewriting the libraries would require an enormous amount of manual work [2].
- Automatically translating Fortran or C libraries into Java. We are aware of two groups that have been working in this area – University of Tennessee [4] and Syracuse University [6]. This approach offers an important long-term perspective as it preserves Java portability, while achieving high performance in this case would obviously be more difficult.
- Manually or automatically creating a Java wrapper for an existing native Fortran or C library. Obviously, by binding legacy libraries, Java programs can gain in performance on all those hardware platforms where the libraries are efficiently implemented. The price to be paid for this clear advantage, however, is the use of native code which breaks the Java security model and does not allow work with applets.

The automatic binding, which we are primarily interested in, has the obvious advantage of involving the least amount of work, thus reducing dramatically the time for development. Moreover, it guarantees the best performance results, at least in the short term, because the well-established scientific libraries usually have multiple implementations carefully tuned for maximum performance on different hardware platforms. Last but not least, by applying the software re-use tenet, each native legacy library can be linked to Java without any need for re-coding or translating its implementation.

While automatic binding is certainly convenient, sometimes the data conversion may impose a bigger performance penalty. As described in section 2 we have addressed several issues potentially contributing to a bigger time overhead of our mixed-language programming approach. As a result of that, our experiments on IBM SP-2 machines have shown a negligible amount of time spent in the binding itself during the execution of Java programs.

One of the primary goals of our approach has been to gain faster execution times by using Java and legacy scientific libraries written in C or Fortran without sacrificing performance from the available highly

optimized native code. The use of the JCI tool clearly extends Java's usefulness and provides rapid solution to the mixed-language interfacing problem, but the JNI-wrapping techniques introduce certain limitations on application portability and mobility. One possible solution to this problem can be achieved by extending the functionality within the boundaries of a metacomputing environment [8].

5. CONCLUSIONS

This article presents a general approach to combine Java and existing code written in Fortran and/or C into mixed-language applications where Java serves as a front-end component for legacy native libraries. We also show that with these existing performance-tuned libraries already available on different platforms and the wrapper interfaces generated by the JCI tool, one can build different kinds of mixed-language software systems for high-performance Java computing in a flexible and elegant way. The JCI tool for automatic creation of interfaces to such libraries (whether for scientific computation or message-passing) plays central role in our mixed-language programming methodology.

In addition to the JCI-generated bindings, other basic components used in our high-performance Java programming methodology include performance-tuned implementations of scientific and communications libraries available on different machines, and a native Java compiler such as IBM's HPCJ. We also believe that our approach is practical in a sense that legacy code is ubiquitous and it would be much too tedious to port all of it to Java. If Java is to gain acceptance as a high-performance language it has to interface with such existing libraries.

Acknowledgments

The author would like to thank Sava Mintchev (University of Westminster) for his significant contribution in the development and implementation of the JCI tool. Special thanks also go to Susan Flynn-Hummel (IBM T.J. Watson Research Center) who invested a lot of time and energy in the experiments with HPCJ, and to Mary Thomas (San Diego Supercomputer Center) for her determination to run the scalability tests on up to 128 IBM SP nodes.

References

[1] Bailey, D., E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga (1994). The NAS parallel benchmarks, Tech-

nical Report RNR-94-007, NASA Ames Research Center. http://science.nas.nasa.gov/Software/NPB/

[2] Bik A. and D. Gannon (1997). A note on native level 1 BLAS in Java, *Concurrency: Pract. Exper.* **9**:11, 1091-1099.

[3] Burns G,, R. Daoud, and J. Vaigl (1994). LAM: An open cluster environment for MPI, in *Proceedings of Supercomputing Symposium '94*, Toronto.

[4] Casanova H., J. Dongarra, and D. Doolin (1997). Java access to numerical libraries, *Concurrency: Pract. Exper.* **9**:11, 1279-1291.

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

[6] Fox G., X. Li, Z. Qiang, and W. Zhigang (1997). A prototype of Fortran-to-Java converter, *Concurrency: Pract. Exper.* **9**:11, 1047-1061.

[7] Getov, V., S. Flynn-Hummel, and S. Mintchev (1998). High-performance parallel programming in Java: Exploiting native libraries, *Concurrency: Pract. Exper.* **10**:11-13, 863-872.

[8] Getov, V., P. Gray, S. Mintchev, and V. Sunderam (1999). Multi-Language Programming Environments for High Performance Java Computing, it Scientific Programming **7**:2, 139-146.

[9] Gordon, R. (1998). *Essential JNI: Java Native Interface.* Prentice–Hall.

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

[11] IBM Corp. (1997). High-performance compiler for Java: An optimizing native code compiler for Java applications. White paper. http://www.alphaWorks.ibm.com/formula/

[12] Mintchev S. and V. Getov (1997). Towards portable message passing in Java: Binding MPI, in *Recent Advances in PVM and MPI* (M. Bubak, J. Dongarra, and J. Waśniewski, eds.), *LNCS* **1332**, Springer-Verlag, Berlin, 135-142.

[13] PARKBENCH Committee (assembled by R. Hockney and M. Berry) (1994). PARKBENCH report-1: Public international benchmarks for parallel computers, *Scientific Programming* **3**:2, 101–146.

DISCUSSION

Speaker: Vladimir Getov

Scott Kohn : What are the memory costs and overheads for using Java for HPC?

Vladimir Getov : This is an interesting but a rather general question. There is a number of issues related to the memory costs and overheads when using Java. First of all, each JVM has its own memory requirements that come in addition to the memory needed by the operating system. Subsequently, the remaining memory available for applications is smaller in comparison to the conventional case of static compilation, including the use of native Java compilers such as HPCJ. For Java application codes in particular running within a JVM, the available memory is defined by the allocated heap size. Tuning the heap size for bigger applications may turn out to be very important in order to utilize the available RAM efficiently. When using the JCI tool, one has to take into account also the JNI overhead and the linking of the specified native libraries at run-time. The wrapper software overhead is relatively very small and can be neglected. In most of the cases the memory costs and overheads vary significantly between different vendors and versions of the Java platform. Therefore, quoting quantitative results should always be accompanied with information about the product, version, release, etc.

Morven Gentleman : Does JCI generate wrappers that can accommodate the need of legacy libraries that require typeless containers, e.g., to support persistent data lifetimes across reverse communication calls?

Vladimir Getov : The JCI tool generates the wrapper code on the basis of mapping between various data types and structures between two given target languages. This mapping can be changed by the user depending on the specific characteristics of the two programming languages and the requirements of the application area. For example, we have used three different mappings so far, but none of them accommodates typless containers. However, typeless containers can be included into a new mapping definition for automatic generation of wrappers to relevant legacy libraries.