Multi-Language Programming Environments for High Performance Java Computing

Vladimir Getov[1], Paul Gray[2], Sava Mintchev[1], Vaidy Sunderam[2]

[1]School of Computer Science, University of Westminster, Harrow HA1 3TP, UK

[2]Emory University, Dept. of Mathematics and Computer Science, Atlanta, Georgia, USA

(Draft)

Abstract

Recent developments in processor capabilities, software tools, programming languages and programming paradigms have brought about new approaches to high performance computing. A steadfast component of this dynamic evolution has been the scientific community's reliance on established scientific packages. As a consequence, programmers of high-performance applications are reluctant to embrace evolving languages such as Java. This paper describes the Java-to-C Interface (JCI) tool which provides application programmers wishing to use Java with immediate accessibility to existing scientific packages. The JCI tool also facilitates rapid development and reuse of existing code. These benefits are provided at minimal cost to the programmer. While beneficial to the programmer, the additional advantages of mixed-language programming in terms of application performance and portability are addressed in detail within the context of this paper. In addition, we discuss how the JCI tool is complementing other ongoing projects such as IBM's High-Performance Compiler for Java (HPCJ) and IceT's metacomputing environment.

1 Introduction

It is generally accepted that computers based on the emerging hybrid shared/distributed memory parallel architectures will become the fastest and most cost-effective supercomputers over the next decade. This, however, makes the search for the most appropriate programming model and corresponding programming environments more important than ever before. Arguably the most serious obstacle to the acceptance of parallel supercomputers is the so-called *software crisis*. Software, in general, is considered the most complex artifact in high-performance computing; since the lifespan of parallel machines has been so brief, their software environments rarely reach maturity and the parallel software crisis is especially acute. Hence, portability and software re-use, in particular, are critical issues in enabling high-performance parallel computing. Application programmers need flexible yet comprehensive interfaces which cover both the shared memory and the distributed memory programming paradigms.

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. Although sockets and the Remote Method Invocation (RMI) interface allow network programming, they are rather low-level to be suitable for SPMD-style scientific programming, and thus, codes based on them would potentially underperform platform-specific implementations of standard communication libraries like MPI. Nevertheless, as a programming language, Java has the basic qualities needed for writing high-performance applications. With the maturing of compilation technology, such applications written in Java will doubtlessly appear. Fortunately, rapid progress is being made in this area by developing optimizing Java compilers, such as the IBM High-Performance Compiler for Java (HPCJ), which generates native codes for the RS6000 architecture [11]. Since the Java language is fairly new, however, it lacks the extensive scientific libraries of other languages like Fortran-77 and C. This is one of the major obstacles towards efficient and user-friendly computationally intensive programming in Java.

In order to overcome the above problems, we have applied a Java-to-C Interface (JCI) generating tool [13] to create Java bindings for various legacy libraries [8]. In this article we show that with the existing performance-tuned libraries already available on different platforms and the multi-language interfaces automatically created by the JCI tool, we can build diferent kinds of multi-language programming environments for high performance Java computing in a flexible and elegant way. In principle, the binding of native libraries to Java has certain limitations though. In particular, for security reasons applets downloaded over the network may not load libraries or define native methods. We also show one possible solution of this problem by using the IceT virtual environment [10]. In this case both processes and data would be allowed to migrate and to be transferred throughout owned and unowned resources, under flexible security measures imposed by the users. We also present some evaluation results, which demonstrate the efficiency of our approach.

2 Automatic Binding of Legacy Codes to Java

Accesing native codes from Java looks not that difficult as Java implementations have a *native method interface* specificaly designed for this purpose. Although the native interface was not part of the original Java language specification [9], and different vendors have offered incompatible interfaces, the *Java Native Interface* (JNI) [12] is now the definitive standard for interfacing native code to Java. Implementing JNI in software development involves more than just dynamic linking to the Java virtual machine. Complications stem from the fact that Java data formats are in general different from those of other languages like C, C++, Fortran, etc. which obviously requires data conversion of both arguments and results in multi-language applications. This conversion is a natural part of the native code in case both parts of a multi-language piece of software are to be written from scratch. For legacy codes, however, an additional interface layer called *binding* or *wrapper* must be written which performs data conversion and other functions if necessary.

Binding a native legacy library[1] to Java may also be accompanied by portability problems as the JNI specification is not yet supported in all Java implementations on different platforms. Thus 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 the additional interface layer automatically plays central role in creating flexible multi-language programming environments.

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. As we mentioned already, different Java native interfaces exist, and thus different code may be required for binding a native library to each Java im-

[1] The most convenient representation of the native code is usually in the form of a library.

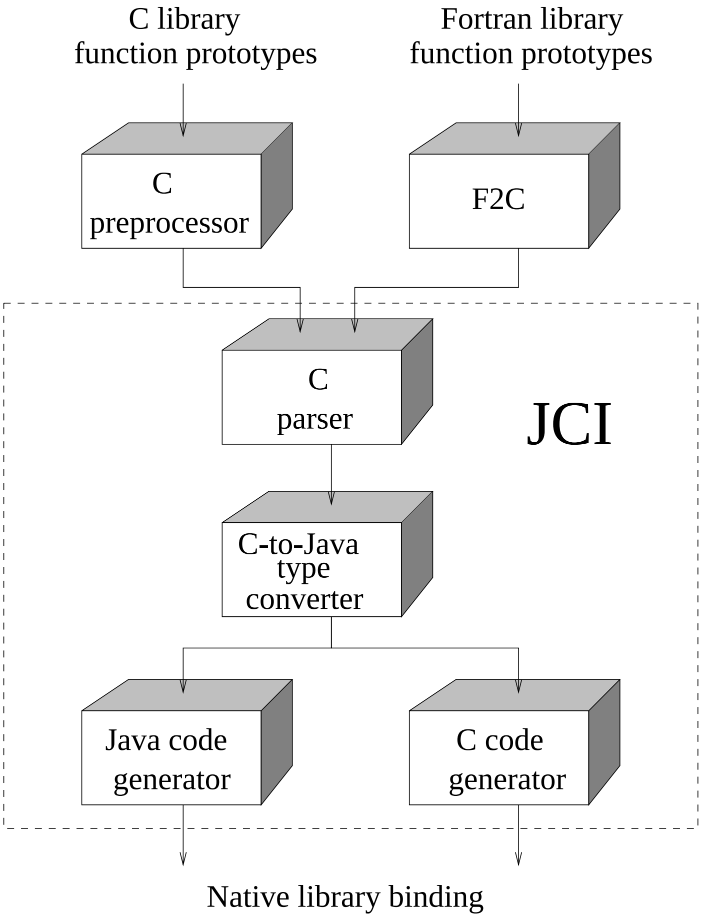

Figure 1: JCI block diagram

plementation. We have tried to limit the dependence of JCI 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 machine simply by providing the appropriate macros.

The JCI tool offers flexible Java bindings for native libraries. For example, by using different library header files as input, we can create bindings for multiple versions of a library, *e.g.* 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. We have created Java bindings for the ScaLAPACK constituent libraries written in Fortran-77: BLAS Level 1–3, PB-BLAS, LAPACK, and ScaLAPACK itself [8]. The C prototypes for the library functions have been inferred by a Fortran–to–C translator *f2c* [6].

While automatic binding is certainly convenient, sometimes there may be a price to pay: the data conversion may impose a performance penalty. For example, some scientific 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 in our ScaLAPACK library bindings. On the other hand, in the Java binding for MPI [13] 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. Multi-dimensional arrays used in MPI as descriptors are relatively small.

3 Environment based on conventional JVM

The initial structure of our programming environment including all basic components is illustrated in Figure 2. The JCI tool takes as input the header file containing the C function prototype declarations of the native legacy library and generates automatically all files comprising the wrapper as required. Then, the bound libraries can be dynamically linked to the Java Virtual Machine (JVM) upon demand and used during the execution. So far we have done experiments with two varieties of the JVM – the Java Development Kit (JDK) for Solaris on a cluster of Sun workstations; and IBM's port of JDK for AIX 4.1 on the SP2.

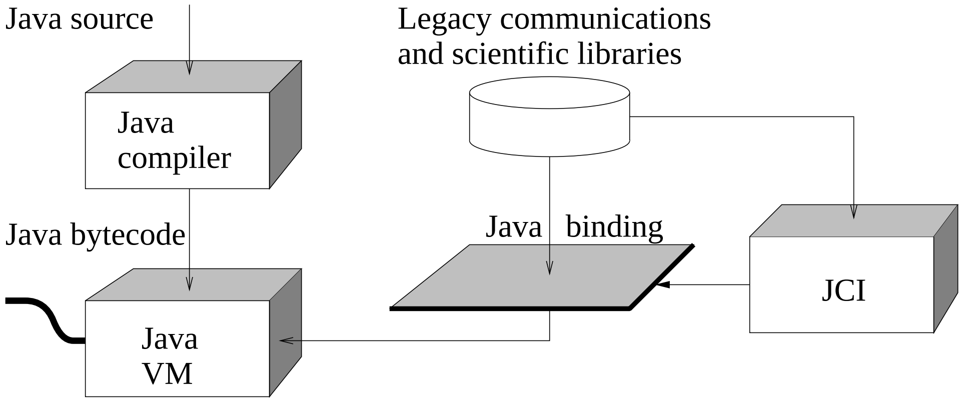

Figure 2: Programming environment using a conventional Java virtual machine

Most JVMs contain a Just-in-Time (JIT) compiler to improve the execution performance. A JIT compiler turns Java bytecode into native code on-the-fly, as it is loaded into the JVM. The JVM then executes the generated code directly, rather then interpreting bytecode, which leads to a significant performance improvement. In this way, the best performance results using the programming environment in Figure 2 can be achieved, but the execution time is still much longer in comparison with similar computations using conventional languages such as Fortran-77 or C and the corresponding compilers. The reason for this noticeable difference is twofold – firstly, the JIT translation adds an extra overhead to the execution time; and secondly, the compilation speed requirements constrain the quality of optimisation that a JIT compiler can perform. Therefore, the performance of this environment is relatively low as there is usually a large imbalance between the efficiency of the performance-tuned implementations of legacy libraries and the rest of the code at execution time.

4 Environment based on native Java compilers

An optimising native code compiler for Java can be used instead of the JVM in order to overcome the above problem. Such a compiler translates bytecode directly into native executable code as shown in Figure 3. It works in the same manner as compilers for C, C++, Fortran, etc. and unlike JIT compilers, the static compilation occurs only once, before execution time. Thus, traditional resource-intensive optimisations can be applied in order to improve the performance of the generated native executable code. In our experiments, we have used a version of HPCJ, which generates native code for the RS/6000 architecture.

The input of HPCJ is usually a bytecode file, but the compiler will also accept Java source as input. In the latter case it invokes the JDK source-to-bytecode compiler to produce the bytecode file first. This file is then processed by a translator which passes an intermediate language representation to the common back-end from the family of compilers for the RS/6000 architecture. The back-end outputs standard object code which is then linked with other object modules and the previously bound legacy libraries to produce native executable code. In this way, our programming environment conforms to the basic requirements for high-performance computing as the experimental results in the next section show.

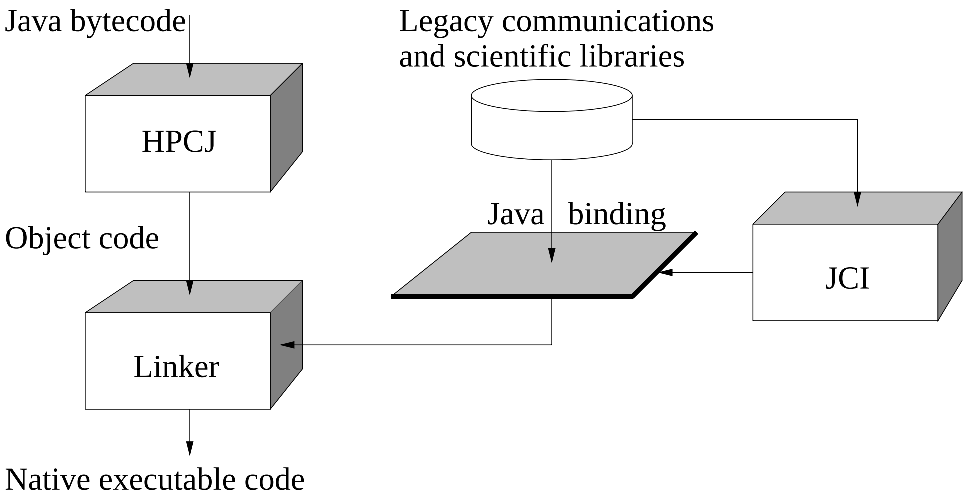

Figure 3: Programming environment using the high-performance compiler for Java which generate native executable code

5 Virtual environment using IceT

Up to this point, the motivation behind mixing languages has been driven by a goal to increase performance of a stand-alone process, in which case the utilization of native codes has significant attraction. Although in most cases the performance improvement when utilizing native codes is substantial, a relatively small price has to be paid. This has already been mentioned, namely the additional overhead incurred in the data translation which is an unavoidable part of the wrapping code when passing between languages. Other costs which are incurred, however, are in *portability* and *platform independence*. These aspects are not as pronounced in the situation of a stand-alone process. However, high-performance computing often entails multi-process computations which span multiple architectures, operating systems, networks, and filesystems. Such an inhomogeneous environment is ideally suited for pure Java processes, where the bytecode representation of the process gives a high-level of assuredness that any component of the multi-process computation can be moved to and executed on any of the resources. This is not the case when one begins mixing languages.

Fortunately, this is not to say that the aspects of portability and platform independence are entirely lost. Such has been the focus of the computational environment of IceT, where Java's bytecode representation is used to give large degrees of portability to mixed-language-based distributed computations. In this sense, the benefits to distributed computations are two-fold: Processes enjoy executional speeds of native-language codes *and* maintain levels of portability similar to pure-Java processes.

For example, Java portability is often recognized in the form of Java-based applets being downloaded over the network and run locally within one's web browser. However, a Java-based process which has been written to take advantage of the JCI binding of MPI for a specific architecture such as Solaris would have little chance in being downloaded to be executed on a Windows95 PC. This lacking aspect of portability is addressed in IceT.

A process' use of native codes is detectable by a judicious investigation of the process' bytecode representation. In IceT, process portability is realized through detecting a process' use of native codes and then supplying the necessary shared library format for the appropriate architecture and operating system. The utility of JCI for automating the wrapping of native libraries is attractive in the sense that a Java-binding of native libraries can be easily generated for multiple platforms. The result of the JCI Java-binding for MPI is a shared library, "`libJavaMPI_MPI.so`" which holds the MPI compilation for the specific architecture and operating system (e.g., Sparc/Solaris, SP2/AIX, etc.). The particular shared library is required in order for the process to successfully execute. In this example, we've used the JCI-generated MPI library to support a distributed message-passing process which is soft-installed onto a remote, pre-configured MPI cluster using IceT.

To accomplish this task, the application programmer writes the programs based upon the JavaMPI binding. Using IceT, JavaMPI calls, and various shell scripts which are located on the remote MPI cluster, the programmer is able to soft-install both the Java-based programming units *and* the supporting JCI-generated MPI library as shown in Figure 4.

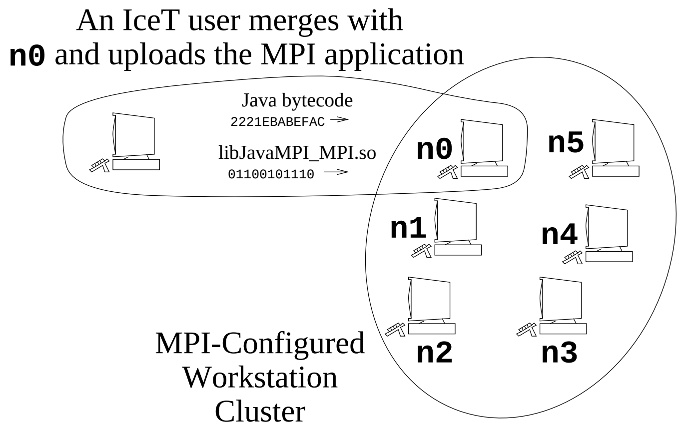

Figure 4: An IceT user (left) merges with a six-node MPI cluster (right) to soft-install and execute an MPI program on the remote cluster.

This example illustrates the additional and often overlooked benefits to Java-wrapping native codes. The MPI program collective in this example enjoys effective communication between processes through the native MPI calls which enables high performance computing. Moreover, the Java-based aspects of the program collective permit a programmer to write and debug MPI applications locally as well as allowing soft-installation of the application onto a larger MPI cluster, or perhaps onto a different MPI configuration such as on an IBM SP2 or Intel Paragon.

6 Experimental results

In order to evaluate the performance of the Java binding to native libraries, we have translated into Java the Matrix Multiplication (MATMUL) benchmark from the PARKBENCH suite [14]. 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. We have run MATMUL on a Sparc workstation cluster, and on the IBM SP2 machine at Southampton University (66MHz Power2 "thin1" nodes with 128Mbyte RAM, 64bit memory bus, and 64Kbyte data cache). The results are shown in Table 1 and Figure 5.

Table 1: Execution statistics for the MATMUL benchmark on the IBM SP2 machine at Southampton University

Problem size (N)	Lang	MPI implementation	No of processors 1	2	4	8	16
Execution time (sec):							
	Java	LAM	—	17.09	9.12	5.26	3.53
1000	F77	LAM	—	16.45	8.61	5.12	3.13
	F77	IBM MPI	33.25	15.16	7.89	3.91	2.20
Mflop/s total:							
	Java	LAM	—	117.0	219.4	380.2	566.9
1000	F77	LAM	—	121.6	232.3	390.4	638.3
	F77	IBM MPI	60.16	132.0	253.6	511.2	910.0

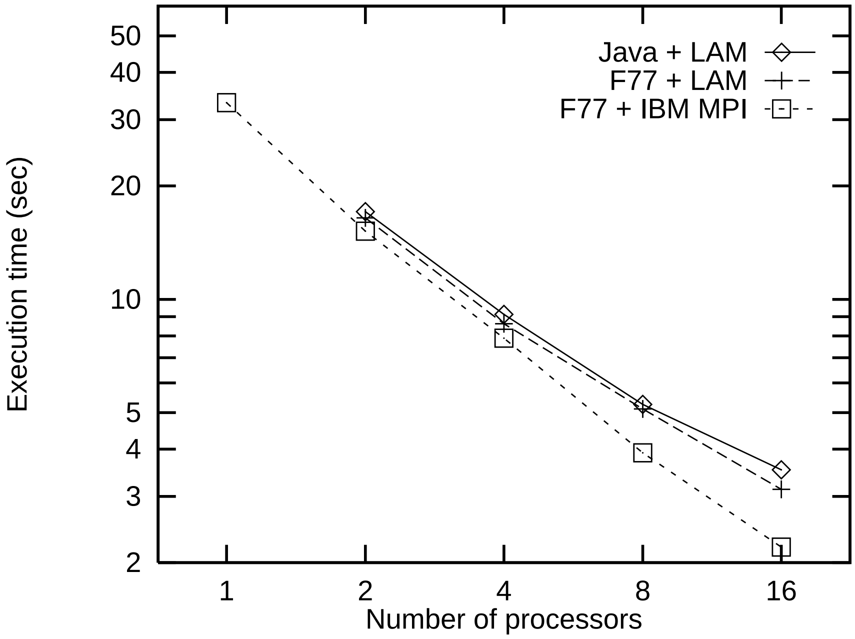

Figure 5: Execution time for the PARKBENCH MATMUL kernel (N = 1000) on the IBM SP2 at Southampton University

Further experiments have been carried out with a Java translation of a C + MPI benchmark – the Integer Sort (IS) kernel from the NAS Parallel Benchmark suite [1], version NPB2.2. The program sorts an array of N integers in parallel; where the problem size (class A) is specified as $N = 8$M. 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 two platforms: a cluster of Sun Sparc workstations, and the IBM SP2 system at the Cornell Theory Center. Each SP node used has a 120 MHz POWER2 Super Chip processor, 256 MB of memory, 128 KB data cache, and 256 bit memory bus. The results obtained on the SP2 machine are shown in Table 2 and Figure 6. The Java implementation we have used is IBM's port of JDK 1.0.2D (with the JIT compiler enabled). The communications library we have used is the LAM implementation (version 6.1) of MPI from the Ohio Supercomputer Center [4]. The results for the C version of IS under both LAM and IBM MPI are also given for comparison.

Table 2: Execution statistics for the NPB IS kernel (class A) on the IBM SP2 at Cornell Theory Center

		MPI imple-	No of processors				
Class	Language	mentation	1	2	4	8	16
		Execution time (sec):					
A	JDK	LAM	—	48.04	24.72	12.78	6.94
	hpj	LAM	—	23.27	13.47	6.65	3.49
	C	LAM	42.16	24.52	12.66	6.13	3.28
	C	IBM MPI	40.94	21.62	10.27	4.92	2.76
		Mop/s total:					
A	JDK	LAM	—	1.75	3.39	6.56	12.08
	hpj	LAM	—	3.60	6.23	12.62	24.01
	C	LAM	1.99	3.42	6.63	13.69	25.54
	C	IBM MPI	2.05	3.88	8.16	14.21	30.35

It is evident from Figure 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 is two times slower than C IS (Figure 6). The explanation is that in MATMUL most of the performance-sensitive calculations are carried out by the native library routines (which are the same for both Java and Fortran-77 versions of the benchmark). In contrast, IS uses a native library (MPI) only for communication, and all calculations are done by the benchmark program.

It is important to identify the sources of the slowdown of the Java version of IS with respect to the C version. To that end we have instrumented the JavaMPI binding, and gathered additional measurements. It turns out that the cumulative time spent in the C functions of the JavaMPI binding is approximately 20 milliseconds in all cases, 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 2.

7 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 in Java. 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:

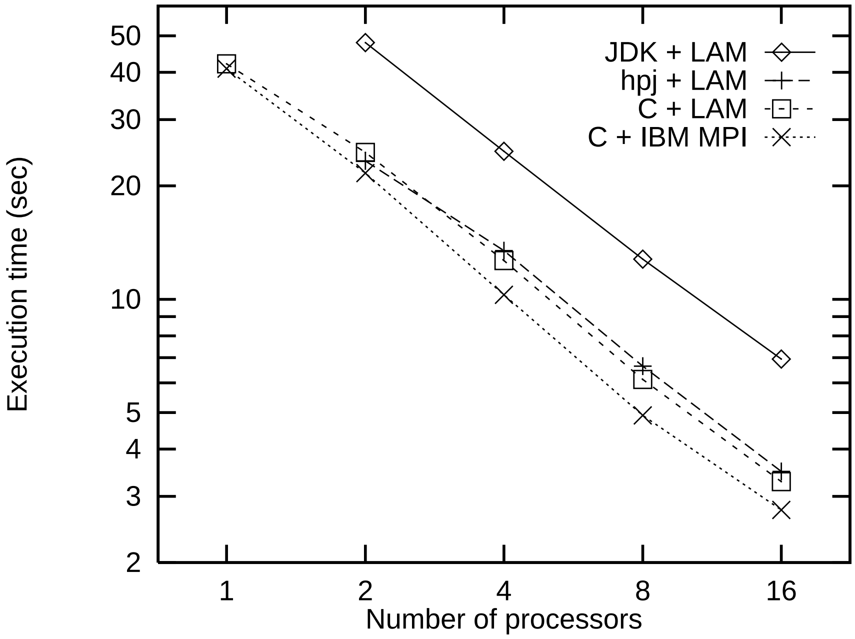

Figure 6: Execution time for the NPB IS kernel (class A) on the IBM SP2 at Cornell Theory Center

- Hand-writing existing libraries in Java. Considering the size of the available libraries 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-77/C library code into Java. We are aware of two research groups that have been working in this area – University of Tennessee [5] and Syracuse University [7]. 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.

- Manually or automatically creating a Java wrapper for an existing native Fortran-77/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 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.

After the initial period when the first Java versions 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, JIT compilers have dramatically improved their efficiency, and are now challenging mature C++ compilers. The developers of HPCJ have adopted the 'native compiler' approach in order to gain faster execution times. A different strategy has been chosen by the authors of Toba [15]. Toba translates Java bytecode into C source code, which is then compiled with the appropriate compiler optimisation flags for high performance. Another advantage of this approach is that it is as portable as any other C software.

8 Conclusions

The JCI tool for automatic creation of interfaces to native legacy libraries (whether for scientific computation or message-passing) improves the portability of such interfaces. In addition to the JCI-generated bindings, the basic components of out high-performance Java programming environment include performance-tuned implementations of scientific and communications libraries available on different machines, and a native Java compiler like IBM's HPCJ.

References

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

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

[3] L. 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. Whaley, ScaLAPACK: A linear algebra library for message-passing computers, in "Proceedings of SIAM Conference on Parallel Processing" (1997).

[4] G. Burns, R. Daoud, and J. Vaigl, LAM: An open cluster environment for MPI, in "Proceedings of Supercomputing Symposium '94," Toronto, http://www.osc.edu/lam.html (1994).

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

[6] 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.

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

[8] V. Getov, S. Flynn-Hummel, and S. Mintchev, High-performance parallel programming in Java: Exploiting native libraries, *Concurrency: Pract. Exper*, to appear (1998).

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

[10] P. Gray and V. Sunderam, The IceT environment for parallel and distributed computing, in "Scientific Computing in Object-Oriented Parallel Environments," Y. Ishikawa, R. Oldehoeft, J. Reynders, and M. Tholburn, eds., *LNCS* 1343 (1997).

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

[12] S. Liang. *The Java Native Method Interface: Programming Guide and Reference.* Addison–Wesley, 1998.

[13] S. Mintchev and V. Getov, Towards portable message passing in Java: Binding MPI, in "Recent Advances in PVM and MPI," M. Bubak, J. Dongarra, J. Waśniewski, eds., *LNCS* 1332 (1997).

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

[15] T. Proebsting, G. Townsend, P. Bridges, J. Hartman, T. Newsham, and S. Watterson, Toba: Java for applications – a way ahead of time (WAT) compiler, in "Proceedings 3rd Conference on Object-Oriented Technologies and Systems (COOTS'97)" (1997).

[16] D. Souder, M. Herrington, R. Garg, D. DeRyke, JSPICE: A component-based distributed Java front-end for SPICE, *Concurrency: Pract. Exper*, to appear (1998).