Aspects of Portability and Distributed Execution for JNI-Wrapped Message Passing Libraries*

Vladimir S. Getov[†] Paul A. Gray[‡§] Vaidy S. Sunderam[¶]

January 25, 2000

Abstract

This paper discusses an approach which aims to provide legacy message passing libraries with Java-like portability in a heterogeneous, metacomputing environment. The results of such portability permits distributed computing components to be "soft-loaded," or "soft-installed" in a dynamic fashion, onto cooperating resources for concurrent, synchronized parallel execution. This capability provides researchers with the ability to tap into a much larger resource pool and to utilize highly-tuned codes for achieving performance. Necessarily, the Java programming language is a significant component. The Java Native Interface (JNI) is used to wrap message passing libraries written in other languages and the bytecode which is generated for the front-end may be analyzed in order to completely determine the needs of the code which it wraps. This characterization allows the pre-configuration of a remote environment so as to be able to support execution. The usefulness of the portability gained by our approach is illustrated through examples showing the soft-installation of a process using an MPI computational substrate and the soft-installation of a process which requires a C-based communication library based upon the efficient multi-cast communication package, CCTL. The examples show that significant gains in performance can be achieved while allowing message passing execution to still exhibit high levels of portability.

1 Introduction

The Java programming language has received a considerable amount of attention since its public introduction and many of the inherent features of the language have since garnered the interest of high-performance computing communities. In looking to Java as a language for high-performance computing though, several preventative barriers inherent to the language have become clear. Executional speed, the present lack of acceptance of an IEEE floating point standard, and the abundance of legacy code are some of the reasons for reluctance of the scientific community to completely accept Java as a high-performance

*Work partially supported through NSF grant ACI-9872167, the University of Northern Iowa's Graduate College, and through the NFF initiative of HEFCE (UK)

†School of Computer Science, University of Westminster, London, UK

‡Contact author: gray@math.uni.edu

§Department of Mathematics, University of Northern Iowa, Cedar Falls, IA 50614

¶Department of Mathematics and Computer Science, Emory University, Atlanta, GA 30322

programming language. These issues have been discussed at length, for example, in [4], [18] and [16] respectively.

During the evolutionary period of the Java programming language, other independent projects have evolved that have taken on the task of extending the definition of a computational environment, such as Globus [3] and Legion [14]. These metacomputing environments share a "computational grid" approach to metacomputing. The scope of such environments has been shown to be well suited for Grand-Challenge problems, but not always suitable for the types of computational situations academic researchers are confronted with today. At this research level, the "Interoperable MPI" (IMPI) [15] standard that is currently under development aims to provide researchers with mechanisms for inter-environmental communication and the chaining together of individual MPI configurations. These projects share the common property of presenting applications with an environment built around high-performance communication layers.

The topic of this paper reflects the ongoing efforts in bringing together separate computational environments to provide researchers with a more light-weight, seamless and flexible environment. The main objective of which, is to provide an environment that can be automatically reconfigured based upon the needs of the application. The target applications for such an environment include collaborative tools, high-performance distributed computations, and other paradigms of computing where applications would benefit from the ability to dynamicdally acquire and move upon new resources.

In this new environment, code portability is essential. While optimized programs written in native languages such as C or Fortran still maintain a significant performance advantage over just-in-time (jit) execution of Java code, they can not exhibit the degree of portability attributable to Java programs through Java's platform-independent bytecode representation. Nonetheless, Java's Native Method Interface (JNI) provides programmers the ability to mix the speed and functionality of native codes with the rapid prototyping of Java. Thus, JNI-wrapped codes provide a programmer with the robustness, speed, and functionality of native methods along with the rapid prototyping of Java at the expense of maintaining total portability and platform independence.

This paper addresses this lucrative aspect of portability of JNI-wrapped code. While truly platform-independent execution is lost, the Java bytecode representation provides a unique quantification of Java-based processes which provides us a mechanism to detect dependence on native codes and to establish the necessary environment prior to its execution on a *remote* host. We show how static (non-executing) Java bytecode can be analyzed on a non-local (remote) host for dependence on native methods. The result of this analysis provides specific information on the requisite environment needed for execution of the process. Once the appropriate environment is established, the process may be executed on a non-local host as naturally as any other.

For illustration, contrast this to the simplest paradigm of a browser loading in Java-based applets over the network. The situation depicted here is that of an application being *uploaded* to a remote host which will load in not only Java bytecode for execution, but also soft-install appropriate shared libraries which contain the JNI-wrapped methods. To facilitate this remote execution of JNI-wrapped native methods, one must minimally be able to load in the static bytecode representation of a Java process and analyze its environmental needs. This bootstrapping of a process includes:

1. detecting the process' dependence on native methods,
2. detecting the process' dependence on other Java classes,
3. obtaining all of the dependency classes and libraries which contain the native method calls,
4. linking the process to the shared libraries containing the native methods,
5. resolving all of the dependency Java classes, and
6. instantiating the process.

These requirements are automated by the IceT environment [12, 13], discussed in the next section.

The IceT mechanism by which the bytecode representation makes this automatic configuration of the environment possible is discussed in section 3. For illustration purposes, examples are given in section 4; one which shows how IceT enables the soft-installation of an MPI process on a remote MPI configuration and a second example which shows how IceT facilitates the soft-installation of CCTL, an efficient multi-cast-based communication package [17].

2 The IceT Model: Potential and Pitfalls

The IceT project began as an investigation into the utility of Java for extending standard models of distributed computing[11]. In the setting where the distributed computing environment consisted of clusters of workstations, these advantages include

- platform independent execution and the Java bytecode representation.

  These features of the Java programming language would ease the requirement that the end user provide a compilation of the program for each possible architecture and operating system present in the pool of computational resources. Such is the case in packages like PVM and MPI for example, where a binary executable must be created for each architecture that the application might run upon.

  The bytecode representation of Java applications would permit an extension of the distributed computing model to include a virtual environment which consists of one's resources and additional, "unowned" resources (such as those of a colleague). This bytecode representation would provide means to load and execute code upon remote resources of more arbitrary architectural makeup.

- inherent security

  Upon suggesting distributed computing environments with "unowned" computational components, the issue of security becomes most prominent. The security aspects of the Java programming language, vis-a-vis the `SecurityManager` class, provide a wide range of configurability in order to govern processes running on unowned resources.

An early prototype of IceT provided users the ability to merge resources, i.e. compose distributed computing environments using local and remote resources. Upon these resources, Java-based processes could be uploaded to any computational resources and instantiated (as an *application*, not an applet).

Processes enrolled in the distributed environment communicate and synchronize using data- and object-packed messages along with identifying message tags per IceT's message-passing API. The message-passing paradigm provided by IceT facilitated distributed parallelism. This early prototype showed most potential for applications in the area of *collaborative computing*; for example, the sharing of collaborative tools amongst researchers, where tools for joint visualization or manipulation of data could be provided and soft-installed on-demand upon resources of a colleague which lacked the tool.

This prototype also showed the significant performance penalty in using Java to build the message-passing paradigm. Each message read in over the network is received in raw form as an array of bytes. The lack of multiple inheritance in Java is prohibitive in the sense that this byte array is not able to be manipulated in a more appropriate object. That is, Java's use of Interfaces instead of allowing multiple inheritance prevents a byte array from being cast into another class object, such as a two-dimensional double-precision array for example. In lieu of being able to cast the bytes of a message immediately into another object, a separate object of the appropriate type must be made available for "absorbing" the message content. Significant amounts of overhead are thus introduced in the instantiation of new objects and memory-to-memory copying of data. Contrast this to the capability of the C programming language where one can simply cast the array of bytes into the appropriate form. Benchmarks focusing on this single aspect of Java as it relates to IceT's unpacking of a message are given in section 4.2. These benchmarks show the significant time lag in unpacking large double-precision arrays when compared to a JNI-wrapped library call which performs the same task using the functionality of C.

Benchmarks contrasting the performance of IceT's Java-based message passing environment to comparable computations using PVM [5] were also made and appear in [13]. The results presented in [13] show the ability of jit-executed IceT processes to achieve speeds close to that of comparable unoptimized, C-based PVM and LINPACK programs in *minimal-message* parallel computations. However, optimized versions of the PVM/LINPACK programs maintained a clear advantage in execution speed.

3 Soft-Installation of Native Codes

The unacceptable performance penalty mentioned in the previous section was the underlying motivation for introduction of an alternative approach in IceT. By incorporating soft-installation of native methods in the IceT environment, modules based upon other languages – more suitable for a specific task – could be used to overcome the inherent limitations of Java-based programs. By simply replacing more efficient and effective native components for the bottlenecks attributable to Java, the resulting program *collective* (the Java classes and native methods) is able to achieve a level of performance and portability not found in any single language.

Facilitating the use of native methods in a distributed heterogeneous environment required the ability to dynamically configure remote environments in order to meet the needs of the processes. By using JNI, a process' use of Java-wrapped native methods can be elicited from a judicious investigation of its static bytecode representation. This investigation produces specific information on the shared library encapsulating the native methods. Once the appropriate shared libraries are identified, IceT has been augmented to obtain and link the Java process to the versions of these shared libraries appropriate for the specific architecture and operating system on which the process is to be instantiated. A brief description of this procedure is outlined below.

In order to detect a Java process' use of native methods prior to instantiating the object from within a JVM, the bytecode representation of the process is analyzed. An analysis of the bytecode prior to instantiation provides two major advantages. The first relates to security. In IceT, the owner of a resource sets the conditions under which soft-installed processes are permitted to run (locally). In doing a static analysis, one can detect use of methods that would be questionable in light of permissions given to soft-installed processes. For example, methods relating to filesystem calls, socket usage, and user shells are able to be detected prior to execution of the code, and creation of the process can be denied. The second advantage of static bytecode analysis relates to configuration of the environment. Specifically, if a process depends upon native methods, the library or libraries containing these methods must be present at instantiation of the object. If the native library is not present an error is generated and all subsequent attempts at linking to that library are ignored by the JVM[1]. Therefore, by performing this a-priori analysis, one can soft-install the requisite components prior to instantiation, and thus insure that the linking steps of the object's instantiation will not fail due to a missing library.

In IceT, remote classes are loaded using an extension of the Java `ClassLoader` class. The IceT class loader has two major amendments to the standard class loader. First, it requires *all* class dependencies to be resolved prior to the application's instantiation rather than loading class dependencies on-demand. Second, the IceT class loader performs a thorough analysis of the application classes' bytecode to determine native library usage and security concerns. The first property is necessitated by the second, for if an application does not load native libraries directly but depends upon classes that load native libraries, the application's successful and secure execution ultimately depends upon the management of the shared libraries. Therefore, determining an application's utilization of native libraries involves a thorough inspection of all of the classes associated with the application.

A standard mechanism for bringing together Java-based programs and programs written in C or C++ is to write a Java-based wrapper class which can be thought of as a proxy used to call the native methods. Typically, the JVM makes the requisite links to the library calls in this Java-wrapper class through a static method of the class, such as:

```
static { System.loadLibrary("cctl"); }
```

This simple mechanism for linking Java to native codes is reflected in the bytecode representation of the Java class. Under the JDK1.2 release of the Sun Solaris Java compiler, "`javac`," this particular section of code is reflected in the bytecode representation as part of the class constructor; the ACC_STATIC method "(<clinit>)."

The actual representation of the above loadLibrary call in the bytecode consists of these six bytes in

[1] This prevents the object from *ever* linking to the native methods — even if the shared library name is elicited from the error and is subsequently introduced to the system. See [9], `java.lang.Runtime.loadLibrary` description for details relating to unresolved linking of native methods.

the (<clinit>) method:

`0x12, 0x0B, 0xB8, 0x00, 0x12,` and `0xB1`.

`0x12` is the 'ldc' instruction to the JVM which takes `0x0B` as its argument. This 'ldc' instruction pushes item `0x0B` (11) from the bytecode's constant pool, which, when constructed, refers to the CONSTANT_String entry, "cctl." The next byte, `0xB8`, is the 'invokestatic' instruction to the JVM which takes `0x00` and `0x12` as arguments. These two arguments form the two-byte reference to the CONSTANT_Methodref index of the constant pool to be invoked, namely the CONSTANT_Methodref at location 18 which comes from (`0x00 << 8 | 0x12`) = `0x12` = 18. The CONSTANT_Methodref located in entry 18 of the constant pool reflects the 'loadLibrary' name_and_type item of the class_index pointing to `java/lang/System`, having the type descriptor `(Ljava/lang/String;)V`[2]. The final byte, `0xB1`, is the JVM 'return' instruction.

While somewhat terse and narrow in focus, this example shows how the use of native libraries may be elicited from a static analysis of the bytecode representation. The constant pool of the associated bytecode must be constructed, the fields of the bytecode parsed, and the actual JVM instructions associated with the methods must be parsed for library-loading methods and their arguments.

Note that this analysis may also allow for restrictions to be placed upon other methods that one might not want a soft-installed process to invoke locally. Methods in `java.io.File`, `java.lang.Runtime` or any extensions which might mask these methods could be detected if one wanted to prevent access to the local filesystem or shells. One could also detect a process' use of these methods during the *execution* of the process using Java's SecurityManager, however this presupposes the successful instantiation of the class. Yet, in order to successfully instantiate a class, the appropriate shared library forms must already be local; hence the need for this static analysis of the bytecode.

While static analysis can detect use of native methods and configure the environment accordingly, it can not characterize all behavior of the process collective. For example, one can use IceT's message-

[2] Java's `Reflection` class could be used to determine loadLibrary calls, but it falls short in being able to determine the actual argument to this call which is needed in order to locate and soft-install the required shared library.

passing facility to send and receive `Serializable` objects, and thus a process could receive a buffer containing a serialized object with methods or subclasses screened for during the static analysis of the soft-installed process. For this reason, monitoring of the local execution of a soft-installed process is supervised using owner-configured SecurityManager classes.

The protocol used to support the trans-location of shared libraries in IceT involves RMI-based shared library repositories. These repositories store shared libraries in multiple formats in directory structures according to the GNU convention "arch-os-name," such as "`x86\Windows 95\FOO.DLL`" or "`i386/Linux/libfoo.so`" for example. When an application's shared library dependence is determined by IceT, the locally-cached shared libraries are searched first, then the local system-level shared libraries are searched. In the event that the shared libraries are not found locally, the IceT repository is queried for the appropriate library for the architecture and operating system. Finally, if the appropriate library is not located at the IceT repository, and if the security protocols permit, the location attempting to instantiate the application is queried for the appropriate shared library format. More details on the aspects of shared library utilization in IceT can be found in [10].

4 Example Implementations

There are two significant motivations for introducing portability of native methods in the IceT environment. One motivation is to utilize the large amounts of established legacy codes as native method components in higher-level applications aimed toward "*metacomputing*" environments. An equally-strong motivation that was mentioned earlier is to gain considerable improvement in the performance of Java-based distributed computing. The examples presented in this section epitomize each of these motivations. The first example shows how the soft-installation mechanism of IceT introduces a higher-level of portability to the rich pool of programs written to utilize MPI. The second example shows how the soft-installation of native methods can be used to enhance performance of distributed Java-based processes by supplanting the Java-based IceT message-passing subsystem with CCTL, an efficient and optimized communication package. In both situations, the portability aspect provided by the Java-based portion of the program collective and the robustness of the native method subsystems produces a symbiotic blending which allows performance and distributed attributes not realizable in either component alone.

Early investigations into the usefulness of wrapping native codes in Java focused on aspects computationally-intense calculations. In [13], the performance of Fortran-based LINPACK routines was compared to comparable programs written in Java. The Fortran LINPACK code was wrapped in Java's JNI and required a thin C-based translation layer. The results shown therein provided insight into the overhead associated with the transition into the JNI-level calculations and the computational performance advantages of native codes over pure Java implementations. The Java-C-Fortran wrapping of the LINPACK routines was performed manually. While somewhat less than ideal, the thin C-wrapper was used for manually re-indexing the multi-dimensional arrays in the transitioning from C to Fortran, and was taken into account in the timings. More recent investigations into the overhead associated with the JNI transition into and out of native code on an IBM SP2 was shown in [7]. Investigations into the overhead introduced by the transition in and out of the JNI layer on other platforms and variations of the Java Virtual Machine implementation are actively being pursued.

While use of native methods is typically associated with increasing the performance of computationally-intense calculations, attributes of *communication* can also receive a significant gain in performance. As depicted in both examples below, the transition from Java into native code permits enhanced communication between processes; MPI's effective communication in the first example, CCTL's efficient transport layer in the second.

4.1 MPI

This example provides an illustration of how IceT's soft-installation mechanism can be used to facilitate a distributed computation which extends beyond resources within the IceT environment. IceT's soft-installation mechanism is used to soft-install a process which will utilize MPI on a pre-configured cluster. While there are ongoing efforts in producing a completely Java-based port of MPI, the established, high-performance MPI libraries written in C are used in order to avoid severe performance penalties on the

MPI portion of the environment.

The binding of the MPI library calls which enables their use by Java-based programs was done using the automated methodologies described in [16]. JNI-wrapping of such a library to Java requires a special binding which permits linking the library dynamically to the Java virtual machine, or linking the library to the object code produced by a stand-alone Java compiler. At first sight it appears that this should not be difficult, but there are some hidden problems however. In particular, in order to call a C function from Java, a formal argument of the C function must be supplied for each 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:

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

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 standard, with the addition of a table of the mapping of C types into Java types [6]. All MPI functions reside in one class (MPI), and all MPI constants in another class (MPIconst). However, there is nothing to prevent us from parting with the MPI-1 C–style binding and adopting a more object–oriented approach by grouping MPI functions into a hierarchy of classes.

In our implementation, MPI is bound to the Java virtual machine for Solaris. The MPI implementation used is LAM (version 6.1) of the Ohio Supercomputer Center [1] and is configured for user-level applications on each system involved. The IceT environment is also pre-configured on each environment.

The configuration of the IceT environment involves specification of local variables that determine class file search paths for IceT applications (over and above Java's classpath), the local search tree for locating IceT shared libraries locally, the location of IceT library repositories, security protocols, and protocols for removing locally-cached soft-installed libraries when the parent application terminates. Presented here, is IceT's soft-installation on a remote host, of a distributed MPI-based process. The MPI environment consisted of a pre-configured MPI cluster comprised of Sun UltraSparc workstations. Functionally, this is achieved as follows:

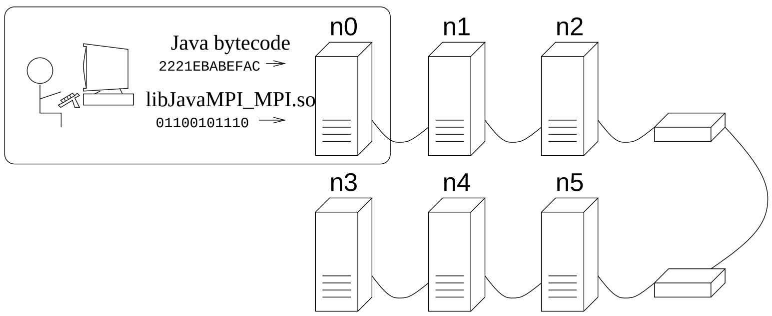

Figure 1: *Using IceT, a user can* merge *with a remote MPI cluster. Java-based processes and any necessary shared libraries may be uploaded to the remote site. A soft-installed MPI-collective is able to run within the remote MPI system.*

1. the user writes code which utilizes the methods in the MPI_Comm and MPI classes of the JavaMPI package. These classes provide the mechanism to interact with and communicate within the MPI environment.

2. on the remote site, MPI is configured several computational resources. On node-0 of these resources, an IceT daemon is started.

3. on the remote site, a shell-script is provided which supplies the mechanism to execute "mpi-run" with the necessary arguments. The name and location of this shell script is provided to the application programmer.

4. the user writes a skeleton class which contains references to the classes to be distributed over the MPI configuration. This skeleton class contains a single execution statement which directs execution

of the "mpi-run" shell script.

5. the user starts an IceT daemon and *merges* with the IceT daemon on node-0 of the remote site.

6. the user requests a spawn of the skeleton class on the remote host, which is node-0 of the MPI cluster.

7. the IceT daemon on node-0 analyzes the bytecode of the MPI skeleton. This analysis detects the references to the classes to be distributed over the MPI cluster mentioned in item (4) and soft-installs them as necessary. These classes are analyzed in-turn, which results in the soft-installation of the appropriate `libJavaMPI_MPI.so` shared library if necessary.

8. the skeleton class is executed as an IceT process. This causes execution of the shell script which in-turn executes the master process of the MPI-based distributed matrix manipulation process.

Using the outline above as a general framework, users are able to write *portable* MPI programs which will execute upon any number of remote MPI configurations.

Note that the IceT program skeleton may be more than just a facilitator for the detection and soft-installation of the requisite components of the distributed MPI run. That is, the "skeleton" that is sent over may be written so as to enroll in both IceT and the MPI computation. In this way, the IceT process could be made aware of the state of the MPI computation, and the original user would be able to manipulate and interact with it.

4.2 CCTL

The use of Java for the scaffolding for the message-passing subsystem of IceT provides several immediate benefits but at a cost. Java provides a unique benefit in the *serialization* of objects. Objects which are "Serializable" are able to be packed into messages as easily as language primitives. However, the lack of pointers in the language prohibits casting between pointer types, which would permit certain optimizations of buffer methods. As a consequence, a large amount of overhead is incurred in allocation of new objects and the memory-to-memory copying of objects to and from buffers.

What this means is that the message-passing substrate of IceT is not optimally suited for passing large messages between tasks. This is somewhat contrary to the demands of distributed high-performance computations. Nonetheless, IceT's soft-installation mechanism and the additional portability of JNI-wrapped native methods can be used to overcome this shortcoming of the language.

The "Collaborative Computing Transport Layer," CCTL, was developed to support multi-user collaborative tools across both local- and wide-area networks as part of the "Collaborative Computing Framework" (CCF) project at Emory University [2]. Using IceT, one can provide an additional dimension to the communication package, namely *portability* of collaborative tools. By writing IceT programs which utilize a JNI-wrapped CCTL library, a user's collaborative tool may be soft-installed upon a colleague's resources for subsequent interaction. Thus, for example, one could share a molecular visualization tool with a colleague using the soft-installation mechanism of IceT and the JNI-wrapping of the CCTL library.

The example presented here shows how the soft-installation mechanism of IceT can be coupled with the portability of JNI-wrapped native methods of CCTL, to achieve high-performance communication across wide-area networks in a multi-user setting. A program written to be portable over an IceT environment is also able to supplant the Java-based IceT message-passing substrate with a JNI-wrapped CCTL library in order to gain effective communication.

Double-Precision Array Size	IceT's Java-based Message-Passing Substrate	JNI-Wrapped CCTL-based Message-Passing Substrate
1	38	6
10	19	3
100	34	3
1,000	250	9
10,000	2538	59
100,000	23662	622

Table 1: *Time (in milliseconds) required to pack, send, and unpack double-precision arrays using IceT's Java-based message-passing substrate and the JNI-wrapped CCTL library.*

The data in Table 1 provides the results of the use of CCTL in this example, and provides motivation for abandoning the Java-based message-passing substrate of IceT when passing large messages amongst processes.

Benchmarks for the overhead associated with the MPI binding have also been performed and the results are given in [8]. The MPI benchmarks give a clear indication of the overhead associated with transitions into and out of the JNI layer (see [8]).

In the CCTL example presented in Table 1, two processes representing a sender and a receiver benchmark times to send and receive double precision arrays over the network. The "`Sender.class`" is spawned on a local host in the IceT environment while the "`Receiver.class`" component is soft-installed along with the appropriate JNI-wrapped CCTL library.

The `Sender` first uses the message-passing substrate of IceT to pack and send double-precision arrays of various lengths which are received and unpacked by the `Receiver`. The total time incurred in packing, sending, receiving, and unpacking the arrays are given in the first column of Table 1.

The `Sender` and `Receiver` then call upon CCTL to create a reliable cctl-"channel" between them. Once the channel is established, the `Sender` utilizes a cctl-based send call to deliver the array to the `Receiver`. The total time incurred in the CCTL transport of the array is given in the second column of Table 1.

5 Conclusions

The JNI-binding of legacy code to Java offers advantages in rapid software development and higher performance but has certain limitations. In particular, mobility of JNI-wrapped codes is typically unsupported due to lack of a mechanism to handle heterogeneous platforms and executable locations. In addition, portability is often partially restricted due to the presence of platform-specific native code. In this paper we describe a solution to this problem by introducing the detection and soft-installation of native codes within the IceT environment.

While JNI-wrapping of native methods is usually presented from the vantage point of increasing computational performance on a single machine, the MPI and CCTL examples in this paper show the benefits of providing Java bindings to communications packages for supporting distributed application programming. By combining the autobinding techniques to provide Java bindings to scientific packages written in other languages and IceT's soft-installation mechanism, application programmers are given the best of both worlds: Applications enjoy enhanced portability and wider accessibility to resources similar to pure-Java applications and are able to derive high-performance levels exclusive to scientific languages such as C.

References

[1] Burns, G., Daoud, R., and Vaigl, J. LAM: An open cluster environment for MPI. In *Supercomputing Symposium '94* (Toronto, Canada, June 1994). Source available at http://www.osc.edu/lam.html.

[2] Chodrow, S., Cheung, S., Hutto, P., Krantz, A., Gray, P., Goddard, T., , Rhee, I., and Sunderam, V. CCF: A Collaborative Computing Framework. *IEEE Internet Computing* (Jan/Feb 2000), 16–24.

[3] Foster, I., and Kesselman, C. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputing Applications* (May 1997).

[4] Fox, G. C., and Furmanski, W. Java for parallel computing and as a general language for scientific and engineering simulation and modeling. *Concurrency: Practice and Experience 9*, 6 (June 1997), 415–426.

[5] Geist, G. A., and Sunderam, V. S. The PVM system: Supercomputer level concurrent computation on a heterogeneous network of workstations. In *Proceedings of the Sixth Distributed Memory Computing Conference* (1991), IEEE, pp. 258–261.

[6] Getov, V., Flynn-Hummel, S., and Mintchev, S. A Programming Environment for High-Performance Computing in Java. *High Performance Computing* (1998).

[7] Getov, V., Gray, P., and Sunderam, V. MPI and Java-MPI: Contrasts and comparisons of low-level commnication performance. In *Proceedings of Supercomputing 99* (Nov. 1999).

[8] Getov, V., Gray, P., and Sunderam, V. MPI and Java-MPI: Contrasts and comparisons of low-level commnication performance. In *Proceedings of Supercomputing 99* (Nov. 1999), The OX Association for Computing Machinery.

[9] Gosling, J., Joy, B., and Steele, G. *The Java Virtual Language Specification*. Addison Wesley, 1996.

[10] Gray, P., and Sunderam, V. Building Distributed Applications Using Multiple Heterogeneous Environments. Submitted to a special edition of Parallel Applications and Architectures, September 30, 1999. Guest editor Dr. Peter Parsons, University of Reading, RG6 6AY, UK.

[11] Gray, P., and Sunderam, V. IceT: Distributed Computing and Java. *Concurrency: Practice and Experience 9*, 11 (Nov. 1997), 1161–1168.

[12] Gray, P., and Sunderam, V. The IceT Environment for Parallel and Distributed Computing. In *Scientific Computing in Object-Oriented Parallel Environments* (New York, Dec. 1997), Y. Ishikawa, R. R. Oldehoeft, J. V. W. Reynders, and M. Tholburn, Eds., no. 1343 in Lecture Notes in Computer Science, Springer Verlag, pp. 275–282.

[13] Gray, P., and Sunderam, V. Native Language-Based Distributed Computing Across Network and Filesystem Boundaries. *Concurrency: Practice and Experience 10*, 1 (1998).

[14] Grimshaw, A., Wulf, W., and French, J. Legion: The Next Logical Step Toward a Nationwide Virtual Computer. Tech. rep., University of Virginia, 1994. T-R Number CS-94-21.

[15] IMPI Steering Committee. IMPI – Interoperable Message-Passing Interface. Draft prtocol version 0.0, currently available at ftp://ftp.nist.gov/pub/hpss/interop/impi-report.current.ps, Oct. 1999.

[16] Mintchev, S., and Getov, V. Automatic binding of native scientific libraries to Java. In *Scientific Computing in Object-Oriented Parallel Environments* (New York, Dec. 1997), Y. Ishikawa, R. R. Oldehoeft, J. V. W. Reynders, and M. Tholburn, Eds., no. 1343 in Lecture Notes in Computer Science, Springer Verlag, pp. 129–136.

[17] Rhee, I., Cheung, S., Hutto, P., and Sunderam, V. Group communication support for distributed collaboration systems. In *Proceedings of ICDCS* (May 1998).

[18] W.Kahan, and Darcy, J. How Java's Floating-point Hurts Everyone Everywhere. Presentation at the ACM Workshop on Java for High-Network Computing, Mar. 1998. document available at www.cs.berkeley.edu/~wkahan/JAVAHurt.pdf.