Towards Portable Message Passing in Java: Binding MPI

Sava Mintchev and Vladimir Getov

School of Computer Science,
University of Westminster, London, UK
http://perun.scsise.wmin.ac.uk/
{s.m.mintchev, v.s.getov}@westminster.ac.uk

Abstract. In this paper we present a way of successfully tackling the difficulties of binding MPI to Java with a view to ensuring portability. We have created a tool for automatically binding existing native C libraries to Java, and have applied the Java–to–C Interface generating tool (JCI) to bind MPI to Java. The approach of automatic binding by JCI ensures both portability across different platforms and full compatibility with the MPI specification. To evaluate the resulting combination we have run a Java version of the NAS parallel IS benchmark on a distributed–memory IBM SP2 machine.

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 even more important than it has been so far. Users need a flexible yet comprehensive interface which covers both the shared-memory and the distributed-memory programming paradigms.

Java provides built-in classes and methods for developing multithreaded programs [16] which are suitable for shared-memory computers. As a programming language, it has the basic qualities needed for writing high–performance applications. With the maturing of compilation technology, such applications written in Java will doubtlessly appear. What is necessary, however, is support for programs on distributed-memory architectures. While sockets in Java provide a low–level interface to communication protocols, a higher–level message–passing interface, such as MPI, is required for creating distributed-memory or hybrid applications.

In our view, a Java MPI binding broadens the appeal of both the interface and the language, by combining the convenience of programming in a high-level language with the potential for high performance. We expect such a combination to be of interest not only to scientific computing, but also, for example, to parallel theorem proving and AI communities.

The binding of MPI to Java amounts to dynamically linking an existing C library to the Java virtual machine. At first sight it appears that this should not

be a problem, as Java implementations support a *native interface* via which C functions can be called. There are some hidden problems, however. First of all, native interfaces are reasonably convenient when writing new C code to be called from Java, but rather inadequate for linking pre-existing C code. The difficulty stems from the fact that Java has in general different data formats from C, and therefore existing C code cannot be called from Java without prior modification.

Linking a C library (*e.g.* MPI) to Java is also accompanied by portability problems. The native interface is not part of the Java language specification [9], and different vendors offer incompatible interfaces. Furthermore, native interfaces are not yet stable and are likely to undergo change with each new major release of a Java implementation[1]. Thus to maintain the portability of MPI libraries (which is after all their defining feature!), one would have to cater for a variety of native interfaces.

2 Binding a native MPI library to Java

In order to call a C function from Java, we have 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:

- 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.

We want to link a large C library — MPI — to a Java virtual machine. Because of the disparity between C and Java data types, we are faced with two options:

1. Rewrite the library C functions so that they conform to the particular native interface of our Java VM; or
2. Write an additional layer of "stub" C functions which would provide an interface between the Java VM (or rather its native interface) and the library.

Software engineering considerations make option (1) a non-starter: it is not our job to tamper with a library supported by others. But option (2) is not very attractive either, considering that a native library like MPI can have hundreds of accessible functions. The solution is to choose (2), and automate the creation of the additional interface layer.

The use of a tool for creating a Java–to–C library interface is illustrated in Figure 1. The *Java–to–C interface generator*, or JCI, takes as input a header file

---

[1] JNI in Sun's JDK 1.1 is regarded as the definitive native interface, but it is not yet supported in all Java implementations on different platforms by other vendors.
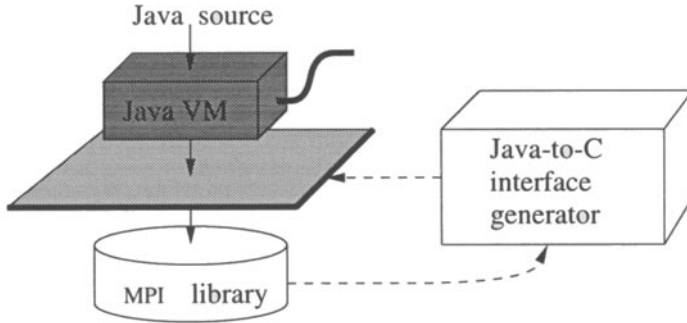
Fig. 1. Binding a native library to Java

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

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

As we mentioned in Section 1, different Java native interfaces exist, and thus different code may be required for binding a native library to each Java implementation. We have tried to limit the implementation dependence of JCI output to a set of macro definitions describing the particular native interface. Thus a library can be re-bound to a new Java machine simply by providing the appropriate macros.

Every MPI library has in excess of 120 functions. The JCI tool allowed us to bind all those functions to Java without extra effort. Since all MPI libraries are standardized, the binding generated by JCI should be applicable without modification to *any* MPI library.

From the application programmer's perspective accessing MPI functions in Java is no harder than it is in C.

Example 1. A small test program using MPI:

```
class TestMPI
{
  public static void main(String args[])
  {
  MPIconst     constMPI= new MPIconst();
  MPI          javaMPI = new MPI();
```

```
ObjectOfJint  argc    = new ObjectOfJint(args.length),
              rank    = new ObjectOfJint(),
              answer  = new ObjectOfJint(0);
constMPI.MPI_Init (argc, args);
javaMPI.MPI_Comm_rank(constMPI.MPI_COMM_WORLD, rank);
if (rank.val == 0) answer.val = 42;
javaMPI.MPI_Bcast (answer, 1, constMPI.MPI_INT, 0,
                   constMPI.MPI_COMM_WORLD);
System.out.println ("My rank is " + rank.val +
                   ", The Answer is " + answer.val);
javaMPI.MPI_Finalize ();
}
}
```

When the program of Example 1 is run in SPMD mode, the root process (whose rank is 0) broadcasts an integer to all other processes. The example illustrates the use of Java objects to simulate C pointers in a type-safe way. The class ObjectOfJint, whose definition has been generated by the JCI tool, contains a single field val of type int. An object, *e.g.* answer, of that class acts as a pointer to an integer, and it can be dereferenced as answer.val.

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[2]. 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.

So far we have bound MPI to two varieties of the Java virtual machine — JDK 1.0.2 [14] for Solaris and for AIX 4.1 [12]. The MPI implementation we have used is LAM of the Ohio Supercomputer Center [2].

3 The IS NAS parallel benchmark in Java

In order to evaluate the performance of Java + MPI we have translated into Java a C + MPI benchmark — the IS program from the NAS Parallel Benchmark suite NPB2.2 [18, 1]. The program sorts in parallel an array of N integers. The size of the array varies with the *class* of the benchmark; for example, for class S $N = 64K$, while for class A $N = 8M$.

We have run the IS benchmark on two platforms: a cluster of Sun Sparc workstations, and an IBM RS/6000 parallel SP2 system at the University of Southampton. Each of the 16 (model 390) nodes of the SP system that we have used has a 66MHz Power2 processor, 128Mbyte RAM (64bit memory bus),

---

[2] For the mapping of primitive types see the documentation of the particular Java implementation; for other types see the documentation of the JavaMPI binding.

32Kbyte instruction cache, and 64Kbyte data cache. The results are shown in Table 1 and Figure 2.

Platform	Class	Language	No of processors				
			1	2	4	8	16
Execution time (sec):							
IBM SP2	S	C	0.21	0.21	0.17	0.16	0.20
		Java	1.84	1.07	0.60	0.39	0.33
		C (NASA results)	29.1	17.4	9.4	5.2	2.8
	A	C	40.52	24.85	13.14	9.30	15.56
		Java	—	132.48	64.70	37.94	33.48
Mop/s total:							
IBM SP2	S	C	3.08	3.13	3.80	4.18	3.21
		Java	0.36	0.61	1.08	1.69	1.96
		C (NASA results)	2.9	4.8	8.9	16.0	29.7
	A	C	2.07	3.38	6.38	9.02	5.39
		Java	—	0.63	1.20	2.21	2.50

Table 1. Execution statistics for the C and Java IS benchmarks

The Java implementation we have used is IBM's port of JDK 1.0.2C (with the JIT compiler enabled), and the MPI library — LAM 6.1. We opted for LAM rather than the proprietary IBM MPI library because the version of the latter available to us does not support multi-threaded programs [11]. Since the benchmark was run on a homogeneous platform the MPI library performed no data conversion.

The NASA Ames Center results quoted in Table 1 are available on-line [18]. Our measurements for the C version of IS differ from those results as we have conducted our experiments on a different parallel machine and MPI implementation.

We have run additional tests in order to analyse the execution time results. First of all, poor scalability is observed, particularly in the C IS timings. Measurements show that the increase in execution time on a larger number of processors is entirely due to two collective MPI operations, MPI_Allreduce and MPI_Alltoallv. The LAM implementation of those operations is based on point-to-point communications, and unlike IBM MPI it does not utilize fully the hardware capabilities. In the case of the Java version of IS the poor scalability of communication is masked by the fact that calculation is slower, and has a larger share of the total execution time.

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
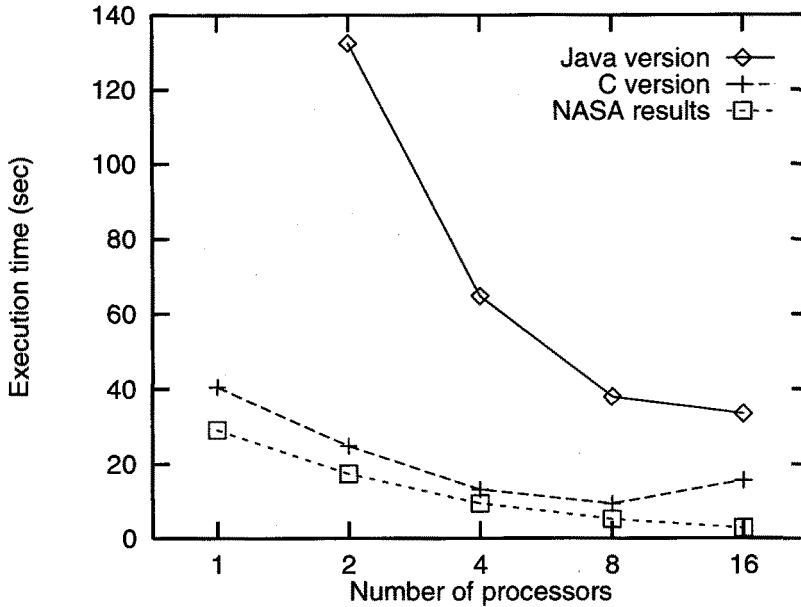
Fig. 2. Execution time for IS class A on the IBM RS/6000 SP system

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.

4 Related work

One of the few other scientific benchmarks in Java that we know of is the sequential Java version of Linpack [5, 6]. It is packaged in a fancy applet that entices web surfers into running the benchmark on their own machines, thus demonstrating the appeal of Java cross–network portability. A number of low–level Java benchmarks have also been created; they are useful for comparing Java implementations, but not for gaining a broader picture of Java performance.

One way of employing Java in high performance computing is to utilize the potential of Java concurrent threads for programming parallel shared-memory machines [13] A very interesting related theme is the implementation on the IBM POWERparallel System SP machine of a Java run–time system with parallel threads [10], using message passing to emulate shared memory. The run–time system would eventually be written in Java with MPI.

A Java-to-PVM interface is publicly available [19]. A Java binding of 25 of the MPI functions has also been written [3, 4] and run on up to 8 processors of Sun Ultra Sparc. In comparison, our binding covers all of MPI; its flexibility

makes it easier to retarget to different versions of the Java native interface, and to Java implementations from different vendors; and it has better portability across hardware platforms.

MPI offers message–passing operations portable over a large variety of machines, but has not to date been bound to many languages. The MPI-1 standard [7] includes bindings for just two languages — C and Fortran 77. A C–linkable language like C++ can use the C binding; nevertheless specific bindings have been proposed [15], and C++ bindings are included in the draft MPI-2 document [8] for both MPI-1 and MPI-2.

5   Conclusions and future work

In this paper we have summarised our work on high performance computation in Java in three major directions:

- Automating the creation of interfaces to native libraries (whether for scientific computation or message passing) and improving the portability of such interfaces with the aid of the JCI tool;
- Creating a Java binding for MPI which is fully compatible with the MPI-1 standard;
- Experimenting with a scientific benchmark in Java — NPB IS — on a distributed–memory machine.

The Java implementation model, and in particular its memory model places certain restrictions on the use of derived MPI data types in Java programs [3]. For instance, multidimensional Java arrays cannot be described as usual by an MPI_Type_contiguous because they are not laid out contiguously in memory. We believe that such restrictions can be alleviated by a special preprocessing and type–checking pass on Java programs which can infer the necessary MPI data types automatically. We have already developed a prototype preprocessor — PMPI [17] — for C and Fortran77, and will be working on its extension to Java next. The Java variant of the PMPI preprocessor will simplify MPI calls by eliminating the redundancy in them, in the way that has already been achieved for C and Fortran.

Apart from extending PMPI, our plans for future work include experimenting with parallel applications in Java on a hybrid shared/distributed–memory machine. This is where we expect the Java thread model combined with explicit message passing to prove truly advantageous.

Finally, we shall continue our work on the JCI tool, enhance the portability of interfaces it generates, and use it to bind some existing native scientific libraries to Java.

Acknowledgments

where the bulk of the experiments presented in this paper were performed; and also to Alan Ogilvie of the Internet Technology Centre at the Hursley IBM UK Lab for answering our questions about the JDK under AIX.

References

1. D. Bailey et al. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, 1994. http://science.nas.nasa.gov/Software/NPB.
2. G. Burns, R. Daoud, and J. Vaigl. LAM: An open cluster environment for MPI. In *Supercomputing Symposium '94*, Toronto, Canada, June 1994. http://www.osc.edu/lam.html.
3. B. Carpenter, Y-J. Chang, G. Fox, D. Leskiw, and X. Li. Experiments with "HP-Java". In *[13]*, 1996.
4. Y-J. Chang and B. Carpenter. MPI Java wrapper download page. 27 March, 1997. http://www.npac.syr.edu/users/yjchang/javaMPI.
5. J. Dongarra. Performance of various computers using standard linear equations software (linpack benchmark report). Technical Report CS-89-85, Computer Science, University of Tennessee, 1997.
6. J. Dongarra and R. Wade. Linpack benchmark – Java version. http://www.netlib.org/benchmark/linpackjava.
7. MPI Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.
8. MPI Forum. MPI-2: Extensions to the Message-Passing Interface. http://www.mcs.anl.gov/Projects/mpi/mpi2/mpi2.html, 1997.
9. J. Gosling, W. Joy, and G. Steele. *The Java Language Specification, Version 1.0*. Addison-Wesley, Reading, Mass., 1996.
10. S.F. Hummel, T. Ngo, and H. Srinivasan. SPMD programming in Java. In *[13]*, 1996.
11. IBM. *PE for AIX: MPI Programming and Subroutine Reference*. http://www.rs6000.ibm.com/resource/aix_resource/sp_books/pe/.
12. IBM UK Hursley Lab. *Centre for Java Technology Development*. http://ncc.hursley.ibm.com/javainfo/hurindex.html.
13. *Workshop on Java for High Performance Scientific and Engineering Computing, Simulation and Modelling*, Syracuse, New York, December 1996. To appear in *Concurrency: Practice and Experience*. http://www.npac.syr.edu/projects/javaforcse.
14. JavaSoft. Home page. http://www.javasoft.com/.
15. Dennis Kafura and Liya Huang. mpi++: A C++ language binding for MPI. In *MPI Developers Conference*, University of Notre Dame, June 1995.
16. D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1996.
17. S. Mintchev and V. Getov. PMPI: High-level message passing in Fortran77 and C. In Bob Hertzberger and P. Sloot, editors, *High–Performance Computing and Networking (HPCN'97)*, pages 603–614, Vienna, Austria, 1997. Springer LNCS 1225.
18. NASA Ames Research Center. *NAS Parallel Benchmarks 2.2 results*. http://science.nas.nasa.gov/Software/NPB/NPB2Results.
19. D.A. Thurman. JavaPVM: The Java to PVM interface. http://www.isye.gatech.edu/chmsr/JavaPVM.