Scientific Programming 11 (2003) 57–66
IOS Press

Mixed-language high-performance computing for plasma simulations

Quanming Lu and Vladimir Getov*
School of Computer Science, University of Westminster, Watford Road, Northwick Park, Harrow HA1 3TP, UK

Abstract. Java is receiving increasing attention as the most popular platform for distributed computing. However, programmers are still reluctant to embrace Java as a tool for writing scientific and engineering applications due to its still noticeable performance drawbacks compared with other programming languages such as Fortran or C. In this paper, we present a hybrid Java/Fortran implementation of a parallel particle-in-cell (PIC) algorithm for plasma simulations. In our approach, the time-consuming components of this application are designed and implemented as Fortran subroutines, while less calculation-intensive components usually involved in building the user interface are written in Java. The two types of software modules have been glued together using the Java native interface (JNI). Our mixed-language PIC code was tested and its performance compared with pure Java and Fortran versions of the same algorithm on a Sun E6500 SMP system and a Linux cluster of Pentium III machines.

Keywords: Java, mixed-language programming, PIC simulation codes, Fortran, high performance computing

1. Introduction

Since its formal introduction in 1995, Java has made extraordinarily rapid progress and has been widely embraced as the programming language of choice for state-of-the-art software development. What makes Java special, but also controversial, is that it is not just a language. Java is also a virtual platform that is not chained to a particular microprocessor architecture or operating system, as the so-called native platforms are. Developers do not have to rewrite or even recompile their Java programs to achieve compatibility with the wide range of currently available computer systems. This saves a lot of work, and gives the users unprecedented freedom to choose the native platforms they want. In order to achieve this, Java requires an extra layer of software, called the Java virtual machine (JVM), to make all native platforms look the same to Java. The Java source code is first compiled into platform independent bytecode, which is then interpreted by the JVM. Therefore, the same bytecode can execute on any platform running a JVM [5]. However, this characteristic of very high portability is also responsible for Javafls performance problems. This is one of the main reasons why the Java platform is generally considered unsuitable for scientific and engineering computing [14].

Nevertheless, as a programming language, Java has some attractive features for high-performance computing. For example, independent threads may be used for programming in Java and then scheduled on different processors by a suitable runtime environment. These built-in mechanisms for parallel processing are well suited for high-performance computing on shared memory machines. Besides these advantages, Java also supports Internet communications and protocols such as sockets and the Remote Method Invocation (RMI) interface. In general, this enables users to solve complex scientific and engineering problems remotely using client-server systems. Recently, with the boost of Internet, such kind of high-performance computing over Internet is becoming more and more popular. A good example of this tendency is the Netsolve project developed at the University of Tennessee [1]. At the same time Java also provides plenty of graphical components such as graphical user interface (GUI) design

*Corresponding author: Vladimir Getov, School of Computer Science, University of Westminster, Watford Road, Northwick Park, Harrow HA1 3TP, UK. Tel.: +44 207 911 5917; Fax: +44 207 911 5906; E-mail: V.S.Getov@wmin.ac.uk.

ISSN 1058-9244/03/$8.00 © 2003 – IOS Press. All rights reserved

facilities including 3-dimensional graphics application programmers interfaces (APIs). In recent years, the attractiveness and suitability of Java for large-scale computing has been also encouraged by the international Java Grande forum. Since 1998, this forum has been coordinating the community efforts to standardize various aspects of Java that support high-performance computing and to ensure that its future development is more appropriate for scientific applications [11].

One way to overcome the performance drawbacks of Java is to use static Java compilers, such as IBM's High-Performance Compiler for Java (HPCJ), which generates optimized native code for the RS6000 and Pentium architectures [17]. Another choice is to use mixed-language programming techniques through JNI where some time-consuming parts of the Java source code are replaced with subroutines written in a high-performance computer language such as Fortran or C [12]. This is what we have done not only for performance reasons, but also because of software engineering considerations. Since Java is a relatively new language, it lacks the extensive scientific libraries that have been developed during the years for other languages like Fortran and C. Using our mixed-language methodology, allows the wealth of existing Fortran and C codes to be reused at virtually no extra cost when writing new applications in Java [4,10].

In this paper, we present our design, implementation, and performance evaluation of both a 2-dimensional and a 3-dimensional mixed Java/Fortran language PIC codes. Some of the details of the PIC algorithm and its parallel version are described in Section 2, while the mixed-language code design and implementation are presented in Sections 3 and 4 respectively. Finally, Section 5 summarizes our performance results collected on two different parallel computing platforms. For the sake of simplicity, most of the following discussions involve only the 2-dimensional PIC code, but similar results for the 3-dimensional version can easily be inferred.

2. PIC simulation skeleton algorithm

The parallel plasma simulation we have started with originally is a skeleton PIC code which was developed by Decyk for benchmarking purposes and evaluation of new algorithms [6]. It uses the electrostatic approximation where magnetic fields are neglected. Therefore, only electrons are being moved during simulation experiments. In this situation, the electrons move in the electric fields which are interpolated from the grid points and can only deposit their charges to grid points. After getting the charge density, the electric fields can be calculated by solving the Poisson's equation using the Fast Fourier Transformation (FFT) method. The periodical boundary condition is applied when implementing this PIC simulation, and the only diagnostics of our code are the particle and field energies. The quadratic spline function is used for the interpolation between grid points and particles, and all the variables are in 64-bit precision.

The physical problem in our particular case is the plasma beam instability where 10% of the particles constitute a beam whose velocity is five times higher than the thermal velocity of the background electrons. Although this code has been deliberately kept as small as possible, it includes the three essential components of a PIC simulation, namely advancing the particles, depositing the charge, and solving the electric fields. The parallel implementation uses *domain decomposition* techniques for porting the PIC code on parallel platforms [9,13]. In this case, different spatial regions are allocated on different processors according to the adopted domain decomposition technique and subsequently the particles are assigned to processors according to the spatial regions they are belonging to. As particles move from one region to another, they are also re-assigned to the processor which is associated with the new region. A one-dimensional domain decomposition as shown in Fig. 1 is used in our code. The global domain is partitioned evenly into several sub-domains (in Fig. 1 the number of sub-domains is 4). Each sub-domain together with its associated electric fields and particles are assigned to a single processor.

3. Mixed Java/Fortran code design

Java is a modern programming language which supports the object-oriented programming (OOP) paradigm. It can *encapsulate* data (*attributes*) and methods (*behaviors*) into *objects* where the data and the methods of an object are intimately tied together. Objects have the property of *information hiding*. This means that although objects may know how to communicate with one another across well-defined *interfaces*, they are not normally allowed to know what is the exact internal implementation of the other objects. Thus, implementation details are hidden with the objects themselves. Java programmers create their own *user-defined types* called classes to instantiate objects.

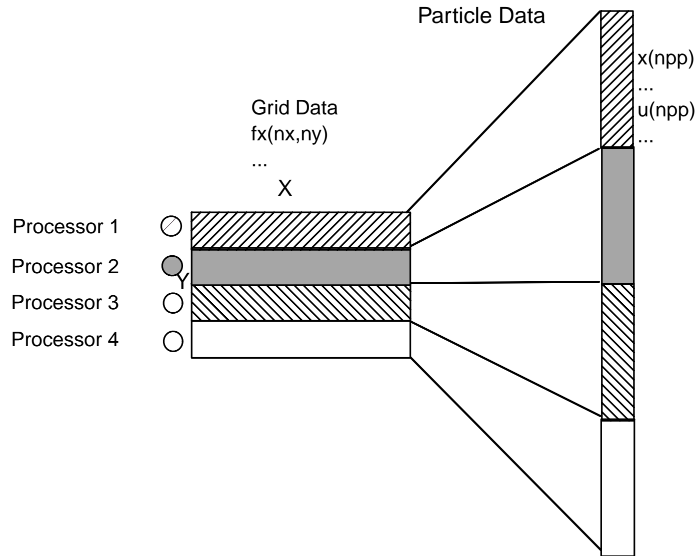

Fig. 1. Grid and particle partitioning.

Each class contains data as well as the set of methods that manipulate this data. The data components of a class are called *instance variables*. In object-oriented languages like Java, one can create a new class from an existing class. The new class inherits the attributes and the behaviors of the existing class. Then programmers can add new attributes and behaviors or override super-class behaviors to customize the class in order to meet their needs. This property is called *inheritance*. In addition, Java also supports *overloading* which means that several methods of the same name can be defined as long as these methods have different sets of parameters. All these capabilities make scientific programs written in Java easier to understand, modify, share, explain and extend. As a result, much more ambitious programming problems can be attacked in a manageable way using the object-oriented approach [7,8,16].

The skeleton parallel PIC simulation code was originally written using Fortran with explicit message passing based on the message-passing interface (MPI) [6]. Recently, it was implemented in Java with the mpiJava library for message passing [15]. The mpiJava package is a reference implementation of the message passing for Java (MPJ) API [3], which was developed as part of the Java Grande forum activities. Using JNI, it is designed and implemented as a wrapper software layer to an existing MPI library [2]. Its purpose for development has been to provide Java programmers with MPI-like functionality for parallel computing which is not part of the standard Java platform.

There are two main kinds of data structures in our PIC simulation code. One is associated with the particles and includes positions and velocities. The other one is associated with the fields and keeps information about the charge density and the electric field in real, complex and Fourier spaces. In line with the two data structures, we can construct two main classes – *plasma* and *field* – both of which extend from the same class named *param*. The *param* class defines some basic parameters such as the number of particles, the number of grid points, etc., which other classes will use.

The *plasma* class declares the particle data as its private data and owns four methods: the *constructor* method is used to initialize the particle positions and velocities; the *push* method interpolates the fields from grid points to particles, then moves particles; the *dist* method assigns the particles who leave this sub-domain to others; the deposit method deposits particle charges to grid points in order to obtain the charge density. The *field* class declares some private data and four methods: the constructor method defines some arrays which will be used by other methods; the *cppfp* method transforms the electric fields from real space to complex space and vice versa; the *fft* method transforms fields from complex space to Fourier space and vice versa; the *pois* method solves the Poisson's equation in Fourier space. Here we don't declare the field data as private data of the *field* class because we will use them for communication between classes as well as inside the class. The basic structure of the parallel PIC code is illustrated in Fig. 2, while the details of its pure Java

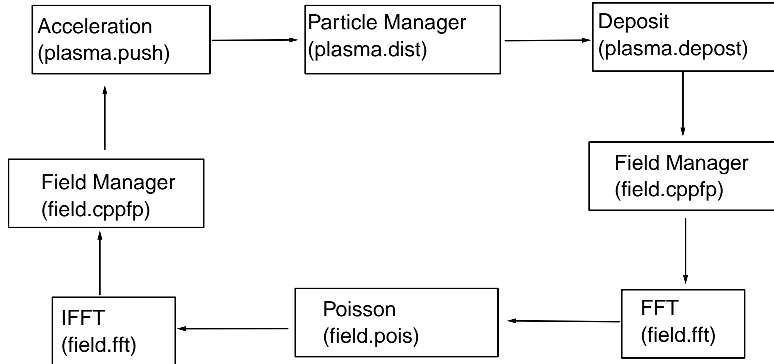

Fig. 2. Structure of the PIC simulation algorithm.

implementation can be found in [15]. It is a main loop containing the following phases: *acceleration, field manager, deposit, field manager, FFT, Poisson solver, IFFT and field manager.* For completeness, their correspondent Java methods are also shown on the same figure. There are actually two *field managers* – one doing field transformations from real space to complex space, and the other one responsible for the reverse process. Both field managers, however, use the same Java method (*cppfp*) with an integer key to distinguish between them. The FFT and IFFT phases correspond to field transformations from complex to Fourier space and from Fourier space to complex space respectively. Again, they use the same Java method (*fft*) with the same mechanism to distinguish between the two as in the *cppfp* method.

Although the PIC code in pure Java is concise and easy to understand and modify, the results show that its performance is only about 20% of that for the Fortran version. Of course, Fortran has been the language of choice for scientific computing for many years, which is one of the main reasons why the scientists are reluctant to do scientific computing in Java. Fortunately, the Java platform provides the JNI API. It allows a Java code that runs within the JVM to operate with applications and libraries written in other languages. The evaluation results for the pure Java version show that the methods *push* and *deposit* consume over 95% of the total runtime on a single processor. Therefore, we decided to replace these two methods with Fortran subroutines. The new mixed-language version first creates the object codes for the two Fortran subroutines with the Fortran compiler, and then invokes these object codes through JNI when running the Java code on the JVM.

4. Implementation details

The execution procedure of our mixed Java/Fortran PIC code is illustrated in Fig. 3. It shows the main steps for compiling Fortran and Java modules separately and the way they are linked at runtime using JNI. Those steps are described below, giving also some more implementation details.

(1) The object codes for the Fortran subroutines *push* and *deposit* are created with the Fortran compiler.
(2) The first two native methods *Jnideposit* and *Jnipush* in the *plasma* class are declared as in Fig. 4, where x and y are the particle positions, vx and vy are the particle velocities, ql is the charge density, and *fxl* and *fyl* are the electric fields. Compiling this Java source file using the *javac* compiler will generate a *plasma.class* file. Then, we run *javah* with the *jni* option on in order to create a header file *plasma.h.*
(3) When writing the Java native method implementation, special care should be taken to interface properly to the Fortran modules. An example for invoking the Fortran subroutine *deposit* through JNI is given in Fig. 5. Note, that it must include the *plasma.h* header file.
Then, using the C compiler we compile the above file while also linking the Fortran object codes to build a shared library called *libparticle.so*.
(4) After completing the above steps, while running on a JVM, one can invoke Fortran subroutines from the main Java loop of the PIC code as shown in Fig. 6.

At first sight, it appears that adopting the above mixed Java/Fortran programming approach should not

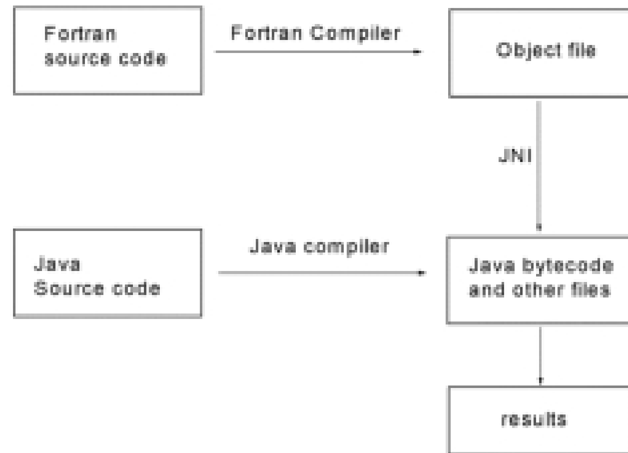

Fig. 3. Block diagram of the mixed Java/Fortran language PIC code.

```
class plasma{
   ......
   public native void Javadeposit(double[] x,double[] y,double[] ql);
   public native double Javapush(double[] x,double[] y,double[] vx,
double[] vy,double [] fxl,double[] fyl);
}
```

Fig. 4.

create any problems. However, complications stem from the fact that the Java data formats are in general different from those in Fortran. When writing mixed-language applications, this obviously requires data conversion of both arguments and results. A couple of peculiarities and difficulties encountered while working on our mixed Java/Fortran code are explained below.

(1) *Multi-dimensional arrays:* The way array layout is organized in Java is completely different in comparison with Fortran, which is a big problem when passing multi-dimensional arrays between Java and Fortran. Therefore, we have changed all multi-dimensional arrays in Java to one-dimension by rearranging the array index to make sure that arrays are passed correctly between Java and Fortran. This also can reduce the runtime overhead which is incurred because multi-dimensional arrays in Java must be relocated in memory in order to be made contiguous before being supplied to Fortran subroutines.

(2) *Array indices:* By default, arrays in Fortran have indices starting from 1, while in Java indices start from 0. When calling existing Fortran subroutines that receive or return an array index, application programmers must be aware of the difference, which they can hide in the interface between Java and Fortran. Of course, this small problem may be overcome by explicitly declaring Fortran arrays with a 0 base, if users are writing new modules in Fortran 90/95.

5. Performance results

In this section, we report benchmark measurements from running both the 2-dimensional and the 3-dimensional mixed Java/Fortran PIC codes on a Sun E6500 and a Linux cluster. We also compare their performance with the corresponding Fortran and Java versions. The Sun E6500 consists of 30×336 MHz Ultra Sparc 2 processors with shared memory. Its operating system during our experiments was Solaris 2.6. We have used Sun's Fortran F90 compiler 2.0 with the following compiler options on: "-O5 -fast − xtarget = ultra2 − xcache = 16/32/1: 4096/64/1". The Java code ran on the JDK1.3 Java platform, while the message passing library for parallel computing was Sun MPI

```
#include <stdio.h>
#include "plasma.h"

JNIEXPORT void JNICALL Java_plasma_Jnideposit(
     JNIEnv *env,jobject thisObj,jdoubleArray x, jdoubleArray
y,jdoubleArray q){
   double *x_f,*y_f,*q_f;
   extern void depost2_();
   x_f=(*env)->GetDoubleArrayElements(env,x,NULL);
   y_f=(*env)->GetDoubleArrayElements(env,y,NULL);
   q_f=(*env)->GetDoubleArrayElements(env,q,NULL);
   depost2_(x_f,y_f,q_f);
   (*env)->ReleaseDoubleArrayElements(env,q,q_f,0);
   (*env)->ReleaseDoubleArrayElements(env,x,x_f,JNI_ABORT);
   (*env)->ReleaseDoubleArrayElements(env,y,y_f,JNI_ABORT);
}
```

Fig. 5.

```
class pic2d{
   static{
         System.loadLibrary("particle");   //load the native library
   }
   public static void main(String args[]){
      ……
      for(int k=0; k<nloop; k++){
         ……
         plasma.Jnipush(x,y,vx,vy,fxl,fyl);
         plasma.Jnideposit(x,y,ql);       //invoke Fortran subroutines
         ……
      }
   }
}
```

Fig. 6.

2.0. The Linux cluster was build out of 16 PC computers, connected with 100 Mb/s fast Ethernet switches. Every computer had a 900 MHz Intel Pentium III processor with 256 Mbyte of memory. The operating system was Redhat Linux 6.2 with the JDK 1.3 Java platform installed, and the Gnu Fortran compiler with the optimization set at level "-O3". The message passing library on the Linux cluster was MPICH 1.2. Both the pure Java and the mixed Java/Fortran versions needed an MPJ interface – hence, the mpiJava1.2 package was installed on both computer systems in order to bind the Java code to the MPI library.

In our experiments, both the 2-dimensional and the 3-dimensional PIC codes have used the same number

Table 1
The total run time in seconds for three versions of the PIC simulation code on a Sun E6500 and a Linux cluster. (a) 2-dimensional code with 32768 grid points and 1310720 particles for 325 time steps; (b) 3-dimensional code with 32758 grid points and 294912 particles for 425 time steps

(a)

Number of	Sun E6500			PC cluster		
processors	Fortran	Java	Mixed	Fortran	Java	Mixed
1	565.46	3849.74	995.92	503.35	1215.02	698.61
2	280.06	1848.26	552.96	248.07	607.24	355.78
4	141.29	921.65	275.87	127.76	317.51	178.27
8	73.39	471.83	138.29	72.61	168.07	97.78
16	44.90	257.07	86.05	47.48	98.05	65.62

(b)

Number of	Sun E6500			PC cluster		
processors	Fortran	Java	Mixed	Fortran	Java	Mixed
1	821.93	5222.12	870.69	547.21	1339.16	614.27
2	388.88	2567.14	458.81	279.05	660.89	330.04
4	169.36	1275.15	231.72	150.39	350.57	184.81
8	97.38	663.68	147.70	89.57	193.58	115.84
16	58.55	368.08	109.96	55.68	108.78	83.72

(32768) of grid points, while their particle numbers are 1310720 and 294912 respectively. Their total run time results excluding the initialization time are shown in Table 1 and plotted in Fig. 7. We ran the 2-dimensional version for 325 time-steps and the 3-dimensional version for 425 time-steps to ensure that the beam instability is fully developed. This is a fair decision, because the initialization phase is not parallelized in order for all of the particles to have always the same initial values regardless of the number of processors. Therefore, the calculated energy and other results during the initialization are always the same. This approach is also useful to uncover subtle bugs.

The results show that the performance of Fortran is about 6 times higher than that of pure Java, but also and more importantly that the mixed Java/Fortran version can highly improve the Java performance. For the 2-dimensional code, the mixed version can increase the Java performance 3–4 times on the Sun E6500 machine and about 1.7 times on the Linux cluster. It can attain about 50% of the Fortran performance on the Sun E6500 and 70% of the Fortran performance on the Linux cluster. For the 3-dimensional code, the mixed version can improve the Java performance 4–6 times on the Sun E6500 and 1.5–2 time on the Linux cluster. The same 3-dimensional mixed version delivers 50%–90% of the Fortran performance on the Sun E6500 and 75%–90% of the Fortran performance on the Linux cluster. When the number of processors increases, the performance improvement achieved by the mixed version decreases. The 3-dimensional mixed-language version demonstrates better this effect, because the Fortran parts of this code consume less and less percentage of the total time when the number of processors increases.

Another aspect of our performance evaluation methodology which deserves mentioning is that, during the process of the beam instability, some processors have more particles than others due to the particle bunch up. This causes load imbalance of about 10% between different processors at runtime, but we do not redistribute the particles in that case as this is a relatively small difference. However, for the total benchmark time of our codes, we report the longest elapsed time on different processors.

We have also compared specifically the runtime overhead introduced by the methods *push* and *deposit* of the class *plasma* across the three versions of our PIC simulation code – Fortran, pure Java, and mixed. From these two methods, we can assess how many floating operations are needed to move one particle during one iteration. Then, the actual performance – i.e. the floating point operations per second rate – can be calculated after the real time spent in these two methods in one benchmarking experiment is known. Measurement results for both the 2-dimensional and the 3-dimensional versions of the code are listed in Tables 2(a) and 2(b) respectively, so that one can draw similar conclusions for each of them. In addition, one can observe that the 3-dimensional mixed version can deliver higher performance improvements than the 2-dimensional one. This is because the calculation/communication ratio of this code is high, resulting in a smaller performance overhead for the data passing through JNI.

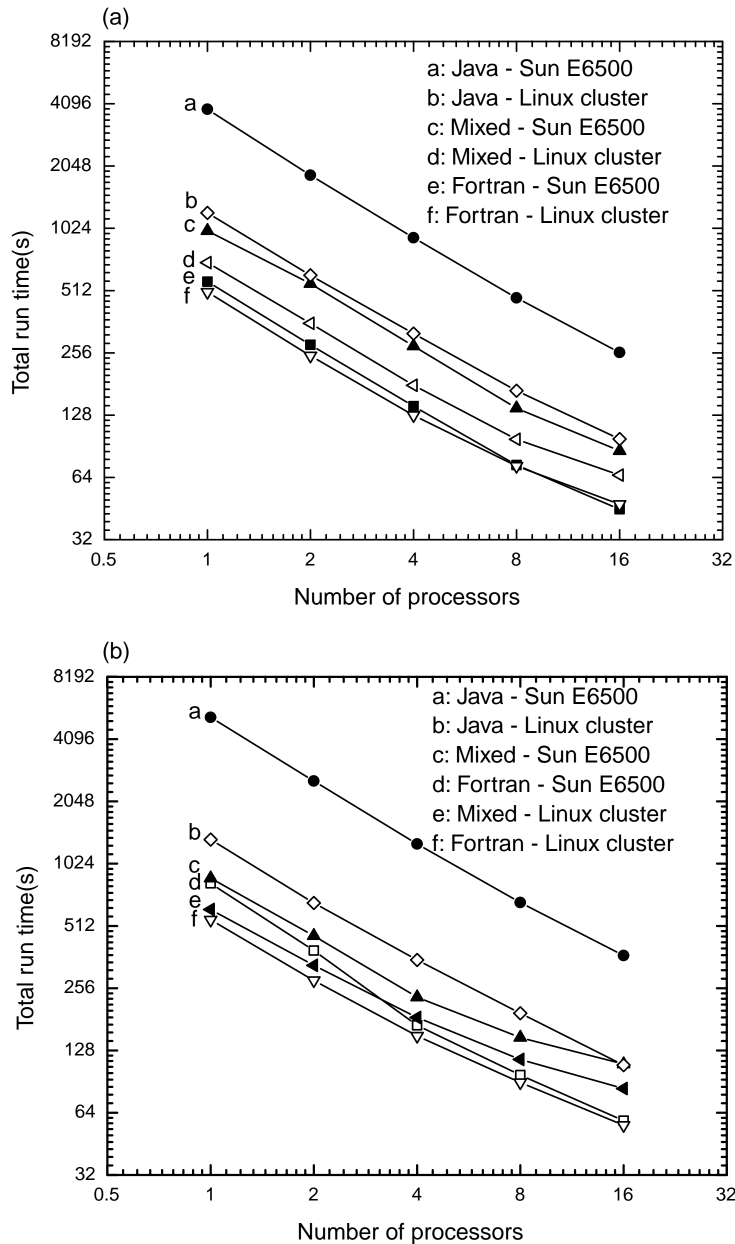

Fig. 7. Total run time in seconds versus number of processors for different versions of the parallel PIC code on a Sun E6500 and a Linux cluster: (a) 2-dimensional code with 32768 grid points and 1310720 particles for 325 time-steps; (b) 3-dimensional code with 32758 grid points and 294912 particles for 425 time-steps.

6. Conclusion

In this paper, we describe a skeleton for object-oriented PIC simulations in Java. Thanks to the object-oriented features of Java, one can easier develop more ambitious PIC simulations based on this skeleton code. We also address the main obstacle on the way to use Java for scientific computing – its relatively low performance. There are two kinds of solutions for this drawback. One is to use native code compilers which produce machine specific executables from Java source code such as IBM's HPCJ. The other one is to use mixed-language programming techniques to replace some of the time-consuming Java methods with subroutines or functions in other high-performance computer languages such as Fortran or C. This is what we

Table 2
The overall performance in Mflop/s for three versions of the PIC simulation code on a Sun E6500 and a Linux cluster. (a) 2-dimensional code with 32768 grid points and 1310720 particles for 325 time steps; (b) 3-dimensional code with 32758 grid points and 294912 particles for 425 time steps

(a)

Number of	Sun E6500			PC cluster		
processors	Fortran	Java	Mixed	Fortran	Java	Mixed
1	95.3	13.9	56.3	107.3	44.0	77.7
2	199.7	29.1	104.8	227.2	88.5	160.3
4	395.5	58.7	211.0	438.2	171.4	324.6
8	786.1	116.0	442.5	845.8	331.3	621.9
16	1527.1	215.3	774.5	1389.6	585.9	771.8

(b)

Number of	Sun E6500			PC cluster		
processors	Fortran	Java	Mixed	Fortran	Java	Mixed
1	62.1	9.4	61.1	93.6	37.8	85.8
2	139.2	19.2	137.2	186.2	78.6	166.3
4	319.8	38.6	284.7	365.5	152.8	315.1
8	591.2	74.2	515.8	660.1	293.1	545.5
16	1200.8	133.8	925.7	1019.4	563.5	795.3

have done and report in this paper. However, both of the above approaches sacrifice the Java portability characteristics across different platforms.

In our work, we have used a PIC plasma simulation code to demonstrate and evaluate the mixed-language approach. In our mixed-language version of the PIC simulation code, we use the most widely used scientific computing language Fortran to implement the most time-consuming particle acceleration and charge deposition subroutines. In turn, these are called from Java through JNI. Finally, the performance of this mixed Java/Fortran PIC code was measured and compared on both single and multiple processors computer platforms. The results show that the mixed Java/Fortran version can highly improve the performance of this application and achieve 70–80% of the pure Fortran performance without much programming efforts. No doubt, more recent and future JVM implementations will further improve these results and make the performance difference negligible.

The mixed-language approach can help to overcome a significant part of Java's low performance problems. Our experiments demonstrate that it can also be used for scientific computing by identifying the hot spots of an application and implement those in Fortran, while using Java for the rest of the code. The price to be paid in that case is that the mixed-language code does not preserve the architecture independence features of pure Java. However, it gives the opportunity to use other attractive Java characteristics such as powerful GUIs, which are becoming more important in modern scientific computing. This would definitely be an advantage in many cases as it helps manipulate and visualize much easier the data obtained from computer simulations. We have already obtained some initial results in this area after adding some basic graphical functions to our Java/Fortran PIC codes.

The same mixed-language methodology can be used to bind Java with the existing scientific libraries written in other computer languages – thus, making those libraries available from Java now. Such an approach would definitely help Java gain acceptance as a scientific computing language well before the key scientific libraries become available in pure Java.

Acknowledgements

The authors would like to thank the University of Wales – Cardiff for the use of their computer systems. Special thanks go also to Bryan Carpenter at Indiana University for his help in the installation of mpiJava1.2 on Sun E6500. This work was financed partly by the HEFCE in the UK under the NFF initiative.

References

[1] D.C. Arnold and J. Dongarra, The NetSolve Environment: Progressing Towards the Seamless Grid, in: *Proceedings of 2000 International Conference on Parallel Processing* (*ICPP-2000*), Toronto Canada, August 21–24, 2000.

[2] M. Baker, B. Carpenter, G. Fox, S.H. Ko and S. Lim, mpiJava: An Object-Oriented Java interface to MPI, in: *Proceedings of International Workshop on Java for Parallel and Distributed Computing,* IPPS/SPDP 1999, San Juan, Puerto Rico, April 1999.

[3] B. Carpenter, V. Getov, G. Judd, T. Skjellum and G. Fox, MPJ: MPI-like message passing for Java, *Concurrency: Practice and Experience* **12**(11) (1999), 1019–1038.

[4] H. Casanova, J. Dongarra and D. Doolin, Java Access to Numerical Libraries, *Concurrency: Practice and Experience* **9**(11) (1997), 1279–1291.

[5] G. Cornell and C.S. Horstmann, *CoreJava,* Sunsoft Press, Mountain View, CA, 1996.

[6] V.K. Decyk, Skeleton PIC codes for parallel computers, *Comput. Phys. Commun.* **87** (1995), 87–94.

[7] V.K. Decyk, C.D. Norton and B.K. Szymanski, How to support inheritance and run-time polymorphism in Fortran 90, *Comput. Phys. Commun.* **115** (1998), 9–17.

[8] D. Flanagan, *Java in a nutshell,* O'Reilly & Associate, Sebastopol, CA, 1997.

[9] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker, *Solving Problems on Concurrent Processors,* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[10] V.S. Getov, A Mixed-Language Programming Methodology for High Performance Java Computing, in: *The Architecture of Scientific Software,* R. Boisvert and P. Tang, eds, Kluwer Academic Publishers, 2001, pp. 333–347.

[11] Java Grande Forum web-site, http://www.javagrange.org/.

[12] S. Liang, *The Java Native Interface: Programmer's guide and specification,* Addison-Wesley, 1999.

[13] P.C. Liewer and V.K. Decyk, A general concurrent algorithm for plasma particle-in-cell codes, *J. Comput. Phys.* **85** (1989), 302–322.

[14] Q.M. Lu and D.S. Cai, Implementation of parallel plasma particle-in-cell codes on PC cluster, *Computer Physics Communications* **135** (2001), 93–104.

[15] Q.M. Lu, V.S. Getov and S. Wang, *Using Java for plasma PIC simulations, Proceedings of IPDPS'03* (*Workshop on Java for Parallel and Distributed Computing*), Nice, France, April 22–26, 2003.

[16] C. Norton, B. Szymanski and V.K. Decyk, Object-oriented parallel computation for plasma simulations, *Communications of the ACM* **38**(10) (1995), 88–100.

[17] V. Seshadri, *IBM high-performance compiler for Java,* AIX-pert Mag., September 1997, http://www.developer.ibm.com/library/aixpert/.