WestminsterResearch
http://www.wmin.ac.uk/westminsterresearch

Using Java for plasma PIC simulations.

Quanming Lu[1,2]
Vladimir Getov[2]
Shu Wang[3]

[1]School of Earth & Space Sciences, University of Science & Technology of China
[2]Harrow School of Computer Science, University of Westminster
[3]ZhuHai Branch, China Netcom Corporation Ltd, P.R. China

Copyright © [2003] IEEE. Reprinted from International Parallel and Distributed Processing Symposium, 2003: proceedings, pp.7-13.

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Westminster's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

The WestminsterResearch online digital archive at the University of Westminster aims to make the research output of the University available to a wider audience. Copyright and Moral Rights remain with the authors and/or copyright owners. Users are permitted to download and/or print one copy for non-commercial private study or research. Further distribution and any use of material from within this archive for profit-making enterprises or for commercial gain is strictly forbidden.

Whilst further distribution of specific materials from within this archive is forbidden, you may freely distribute the URL of WestminsterResearch. (http://www.wmin.ac.uk/westminsterresearch).

In case of abuse or copyright appearing without permission e-mail wattsn@wmin.ac.uk.

Using Java for Plasma PIC Simulations

Quanming Lu[1,2], Vladmir Getov[2] and Shu Wang[3]

[1]School of Earth and Space Sciences, University of Science and Technology of China,
Hefei, Anhui 230026, P. R. China, Email: qmlu@ustc.edu.cn

[2]School of Computer Science, University of Westminster,
Watford Road, Northwick Park, Harrow HA1 3TP, U.K., Email: V.S.Getov@wmin.ac.uk

[3]ZhuHai Branch, China Netcom Corporation Ltd.,
ZhuHai, Guangdong 519020, P. R. China, Email:wangshu@china-netcom.com

Abstract

Plasma particle-in-cell (PIC) simulations model the interactions of charged particles with the surrounding fields. This application has been recognized as one of the grand challenge problems facing the high-performance computing community due to its huge computational requirements. Recently, with the explosive development of Internet, Java is receiving increasing attention and is thought as a potential candidate for high-performance computing. In this paper, we present our approach to developing 2- and 3-dimensional parallel PIC simulations in Java. We also report the execution times for both versions from performance experiments on a symmetric multi-processor (Sun E6500) and a Linux cluster of Pentium III machines. Those results are also compared with benchmark measurements of the corresponding Fortran version of the same algorithm.
Keywords*: Java, PIC simulations, Benchmarking, Message passing, High performance computing*

1 Introduction

A plasma PIC code follows the orbits of particles in the surrounding fields which are calculated self-consistently from the charge and/or current densities incurred by these same particles, where the particles can be located anywhere within the simulation domain but the surrounding fields are defined only on discrete grid points. The basic PIC algorithm consists of an initialization phase followed by four processing phases which are repeated many times: (1) the gather phase interpolates force fields from the grid points to each particle; (2) under the influence of these force fields the particle push phase updates each particle's orbit; (3) then the new charge and/or current densities on the grid points can be deposited from particles by the scatter phase; and (4) at last the field solver phase solves appropriate equations to obtain the surrounding fields. The PIC simulation has long been used by scientists to study the nonlinear kinetic problem in space and laboratory plasma physics [4] with a wide spectrum of implementations using various programming languages on different platforms [1, 14].

Recently, with the development of computer technology, the assumption that high performance computing will be done primarily on specialized supercomputers is questioned increasingly. The rapid progress in performance and connectivity of ordinary workstations and PCs make it look equally possible that the future of the parallel computing will also be on local area networks (LAN) [3] or even the Internet [6]. The growth of the Internet offers the world a high performance massive computational power at very little cost. Programs may be written to take advantage of resources based in logically and geographically different locations without change of their existing infrastructure. However problems arise because of the diversity of operating systems, CPUs and networks involved. To overcome these problems, a paradigm must be created which makes the heterogeneous environment opaque to programmers. Ideally the programmers would create a software application which could be run on a cross-platform metacomputing environment, that is, the environment may be comprised of multi-processor shared memory machines, or a network of workstations, or both.

The development of Java has seen the above possibility brought a step closer. Java source code is first compiled into platform independent bytecode, which is interpreted by

IEEE COMPUTER SOCIETY

a Java Virtual Machine (JVM), so the same bytecode can be run on any platform. Besides this, Java is also an object-oriented language. It is simple and efficient, providing support for various features such as multithreading and Internet communication and protocols. All of these make Java a natural language for network computing, hence making it potentially attractive to scientific programmers hoping to harness the collective computational power of networks of workstations, PCs or the Internet [11]. And the attractiveness of Java for scientific computing has being encouraged by bodies like Java Grande. The Java Grande Forum has been set up to coordinate community efforts to standardize many aspects of Java, and to ensure that its future development will be more appropriate for scientific programmers [10]. One thing that should be done is to obtain experience in how to implement scientific computing in Java and to benchmark its performance. Several suites of benchmark tests have been developed to measure and compare the performance of Java [8, 9], but most of them include kernel-level benchmarks only.

In this paper, we implement two- and three-dimensional PIC simulation codes in Java, which are more realistic benchmarks. The message passing library we used is mpiJava, which provides a Java interface to the widely used Message Passing Interface (MPI). In Section 2, the details of the PIC simulation codes and their parallel algorithms are described. Section 3 discusses the implementation of the object-oriented PIC simulation codes in Java. Finally, the performance of the codes on the Sun E6500 is presented. For the sake of simplicity, most of the following discussions involve only the two-dimensional PIC code. The three-dimensional code can be easily inferred.

2 PIC simulations

There are two main techniques to parallelize PIC codes on parallel platforms. The first is particle decomposition, which assigns evenly the particle population to processors while replicating the whole spatial domain on each processor. The second is domain decomposition. In this algorithm, different processors are allocated to different spatial regions and particles are assigned to processors according to the spatial regions they belong to. As particles move from one region to another, they are assigned to the processor which is associated with the new region. The details including advantages and disadvantages of these two algorithms can be found in [16].

The parallel plasma PIC simulation code we used is a skeleton PIC code which was developed by Decyk as a testbed where new algorithms can be developed and benchmarked [5]. It was originally written using the Fortran language with a message-passing library, and it has been implemented with other parallel programming models [12].

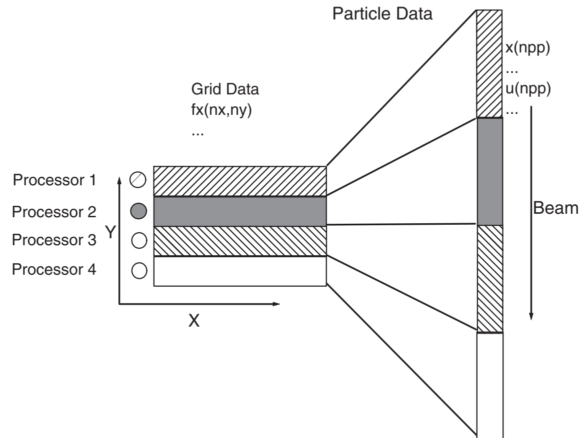

Figure 1. Grid and particle partition.

The code 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 interpolated from the grid points and only deposit their charges to grid points. After getting the charge density, Poisson's equation can be solved with Fast Fourier Transformation (FFT) methods on grid points for the electric potential. Then the electric field can be obtained from the electric potential. Periodic boundary conditions are used when implementing this PIC simulation, and the only diagnostic in this code is particle and field energy. A quadratic spline function is used for the interpolation between grids and particles, and all the variables are in 64 bit precision. The physical problem in this code is beam plasma instability where 10% of the particles are beam particles whose beam velocity is five times the thermal velocity of the background electrons.

Although this code has been deliberately kept minimal, it includes all the essential pieces for a complete PIC simulation – advancing particles, depositing charge and solving fields, such that more complicated PIC algorithms can be easily developed based on this skeleton. A one-dimensional *domain decomposition*, as shown in Figure 1, is used in our code [5, 13]. The domain is divided evenly into several subdomains (in the figure the number of subdomains is 4), and each subdomain with its associated electric fields and particles is assigned to the corresponding processor. In order to provide higher communication burden for benchmarking reasons, the streaming particles are moving normally to the partition surface.

IEEE COMPUTER SOCIETY

3 PIC codes in Java

Java is an object-oriented language. It can *encapsulate* data (*attributes*) and methods (*behaviors*) into *objects*. The data and methods of an object are intimately tied together. *Objects* can communicate with one another across well-defined interfaces without knowing how other objects are implemented – usually implementation details are hidden within the objects themselves. Java programmers create their own user-defined types called *classes* to instantiate *objects*. Each class contains data as well as the set of methods that manipulate the data; the data components of a class are called *instance variables*. Java programmers can also create a new class from an existing class. The new class inherits the attributes and behaviors of an existing class, then adds attributes and behaviors, or overrides the superclass behaviors to customize the class to meet their needs. This property is called inheritance.

The entire code in our plasma PIC simulations is written in Java, except for the low-level interface to the message passing library. Here, we choose to use mpiJava, which is a Java wrapper interface to existing MPI libraries [2]. Its purpose for development has been to provide Java programmers with the traditional functionality of MPI through a Java interface to legacy MPI libraries. This tool enables communication to the underlying MPI library by using the Java Native Interface (JNI) API.

The basic structure of the skeleton PIC code is illustrated in Figure 2. It consists of *initialization* followed by four parts which are *field manager*, *field solver*, *acceleration* and *deposit*. The details of these parts are described as follows:

(1) Initialization: Give each particle its initial position and velocity, calculate initial charge densities on grid points, and initialize some parameters to be used later.

(2) Field Manager: In order to use the fast Fourier transformation (FFT) more efficiently, we change a real sequence into complex sequence by combining every two neighboring real variables into one complex variable (the first is the real part, and the other is the imaginary part) before starting the FFT transformation. The procedure for the inverse Fourier transformation (IFFT) does just the opposite. In addition to this, the interactions between the fields and the particles are not confined to one subdomain, when moving particles or depositing the charges of particles to grid points in one subdomain, the grid points of its neighbors must be used. They are the one uppermost row of the grid points in the beneath subdomain and two lowest rows of the grid points in the upper subdomain, so the field array of the real space must keep three extra rows of grid points which are called *guard points*. Their correspondences in complex space do not have *guard points*. This part transforms the fields from real space to complex (before doing FFT), or vice versa (after doing IFFT).

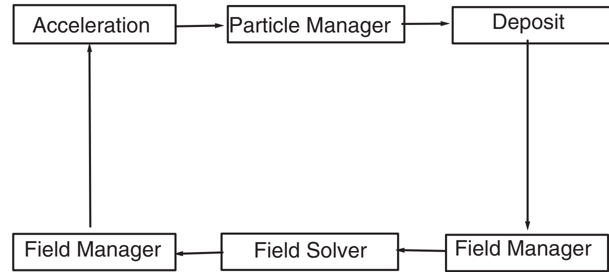

Figure 2. Structure of our skeleton PIC codes.

(3) Field Solver: First change the charge density from complex space to Fourier space, then solve the Poisson equation and get the electric fields in Fourier space, and finally transform fields into complex space with inverse fast Fourier transformation(IFFT). The fields in complex space can be changed to real space by *field manager*, and then used by *acceleration*. When implementing FFT/IFFT in parallel, we first transform in one coordinate direction, then transpose the array, and finally finish the FFT/IFFT in the other coordinate direction.

(4) Acceleration: First interpolate the fields from grid points to particles, and then update the position and velocity of each particle. Then assign the particles that leave this subdomain to corresponding processors.

(5) Deposit: Scatter the charge of each particle to the grid points, and then obtain charge density on grid points.

There are two main kinds of data structures in the PIC code. One kind is associated with particles; it consists of x, y, u, v which are particle positions and velocities in the coordinate directions. The other is associated with fields, it includes q, fx, fy, qc, fxc, fyc which correspond to charge density, electric field in the coordinate directions in real and complex spaces. Because Java doesn't support complex variables, we use one double array with two elements to represent one complex variable, the two elements being the real and imaginary parts of the complex variable, respectively.

If implemented using Fortran or C, each part of the basic structure above corresponds to one subroutine or function. But in Java the fundamental element is the *class* which encapsulates data and methods together. Taking into account the fact that the PIC code has two main data structures, we construct two main classes which extend from one class named *PICSimulator*. The two main classes are named *plasma* and *field*. And the *PICSimulator* class defines some parameters which other classes will use, such as particle number, grid number, etc., it is as follows:

```
class PICSimulator{
   public static final int nx=...,ny=...,nt=..., nvp=...;
```


IEEE COMPUTER SOCIETY

```
  ......
}
```

where *nx* and *ny* are grid numbers in X and Y directions respectively, *nt* is the total particle number on all subdomains and *nvp* is the number of processors used in our simulations. Of course we must define many other parameters which are not listed in the example.

The *plasma* class declares x, y, u, v as its private data and owns three methods: the constructor method used to initialize the particle positions and velocities, the *push* method corresponding to the *acceleration* part which moves particles, and the *depost* method corresponding to the *deposit* part which deposits particle charges to grid points. Its structure is as follows:

```
class plasma extends PICSimulator{
   ......
   private double[] x=new double[npmax], y=new double[npmax],
                    u=new double[npmax], v=new double[npmax];
   public plasma(int myRank) {...}
   public void push(double fx[][], fy[][], dt) {...}
   public void depost(double q[][]) {...}
}
```

where *dt* is time interval, *myRank* is the rank (unique identifier) of the current process returned from the mpiJava library, and *npmax* is the maximum number of particles which subdomains can contain.

In *push* method, it first moves the particles in every subdomain, defines some arrays which contain the particles which will move to other subdomians, then use *mpiJava* library to move these particles, it likes:

```
public void push(double fx[][], double fy[][], double dt) throw
MPIException{

   double[][] sbufr=new double[...][...],sbufl=new double[...][...];
   double[][] rbufr=new double[...][...],rbufl=new double[...][...];
   ......
   for(int j=0; j<npp; j++){
   ......   //move particles and define sbufr and sbufl
   }
   ......
   MPI.COMM_WORLD.Send(sbufr,0,...,MPI.DOUBLE,...,...);
   MPI.COMM_WORLD.Recv(rbufl,0,...,MPI.DOUBLE,...,...);
   ......
   MPI.COMM_WORLD.Send(sbufl,0,...,MPI.DOUBLE,...,...);
   MPI.COMM_WORLD.Recv(rbufr,0,...,MPI.DOUBLE,...,...);
   ......
}
```

where *npp* is the actual number of particle in corresponding subdomain, *sbufr* and *sbufl* are used to contain the particles which move to other subdomains,and *rbufr* and *rbufl* are defined to receive the particles from other subdomains.

The *depost* method deposits particle charges to grid points, the structure is as follows:

```
public void depost(double q[][]){
   for(int j=0; j<npp; j++){
   ......     //deposit particle charges to grid points
   }
}
```

In the *field* class, some private data and three methods are declared: the constructor method defines some arrays which will be used by other methods, and the *cppfp*, *fsolver* methods correspond to the parts *field manager*, and *field solver* respectively. Here we do not declare the field data as private data in the *field* class because we will use them for communication between classes or inside the class. The structure is like:

```
class field extends PICSimulator{
   ......
   public field(int myRank) {...}
   public void cppfp(int isign, double f[][], fc[][][]) {...}
   public void fsolver(double fc[][][],fxc[][][],fyc[][][]) {...}
}
```

Where *isign* is a key. In the *cppfp* method it is used to decide whether field transformation is from real space to complex space, or vice versa. Its structure is as follows:

```
public void cppfp(int isign, double f[][], double fc[][][]) throws
MPIException{
   double[][] sbl=new double[...][...],sbr=new double[...][...],
              rbl=new double[...][...],rbr=new double[...][...];
   if(isign==-1){
     ......
     MPI.COMM_WORLD.Send(sbl,0,...,MPI.DOUBLE,...,...);
     MPI.COMM_WORLD.Recv(rbr,0,...,MPI.DOUBLE,...,...);
     ......
     MPI.COMM_WORLD.Send(sbr,0,...,MPI.DOUBLE,...,...);
     MPI.COMM_WORLD.Recv(rbl,0,...,MPI.DOUBLE,...,...);
     ......
   }
   if(isign==1){
     .....
     MPI.COMM_WORLD.Send(sbl,0,...,MPI.DOUBLE,...,...);
     MPI.COMM_WORLD.Recv(rbr,0,...,MPI.DOUBLE,...,...);
     ......
     MPI.COMM_WORLD.Send(sbr,0,...,MPI.DOUBLE,...,...);
     MPI.COMM_WORLD.Recv(rbl,0,...,MPI.DOUBLE,...,...);
     ......
   }
}
```

where the arrays *sbl*,*sbr* and *rbl*,*rbr* are used to send and receive the data on guard points.

The method *fsolver* is to solve the possion equation with fast Fourier Transformation, the structure is as follows:

```
public void fsolver(double fc[][][],fxc[][][],fyc[][][]){
   double[][][] qt=new double[...][...][...],
   fxt=new double[...][...][...],fyt=new double[...][...][...];
   ......
   isign=-1
   fft(isign,qc,qt);
   pois(qt,fxt,fyt);
   isign=1
   fft(isign,fxc,fxt);
   fft(isign,fyc,fyt);
   ......
   {......}   //define methods fft and pois
 }
```

Where *isign* is the key , it determines to implement FFT or IFFT with the same method. *qt* and *fxt fyt* are the charge density and electric fields in Fourier space.

The main procedure of the PIC code is described as follows:

```
class plasma extends PICSimulator{
   public static void main(String[] args){
      double[][] q,fx,fy;
      double[][][] qc,fxc,fyc;
      ......
      plasma plasma =new plasma(myRank);
      field field=new new field(myRank);
      ......
      for(int k=0; k<nloop; k++){
         isign=-1;
         field.cppfp(isign,q,qc);
         field.fsolver(qc,fxc,fyc);
         isign=1;
         field.cppfp(isign,fx,fxc);
         field.cppfp(isign,fy,fyc);
         plasma.push(fx,fy,dt);
         plasma.depost(q);
        }
    }
}
```


IEEE COMPUTER SOCIETY

where the *nloop* variable defines the number of iteration.

The original Fortran code is written in Fortran 77 style, and its main procedure consists of several subroutines. Inside the Java methods, it is easy to write Java codes from their Fortran counterparts(subroutines), except for two peculiarities: (1) in our Java code we use one double-precision array with two elements to represent one complex variable from the Fortran code; (2) normally Fortran arrays have indices starting from 1 (this is the situation in our Fortran code), while Java indices start from 0. For the communications among processors, we use a standard MPI mode with blocking *send* and *receive* for message passing.

4 Performance Results and Discussions

In this section, we present the performance results of the 2D and 3D plasma PIC codes both in Java and Fortran on a Sun E6500 and a Linux cluster. The Sun E6500 consists of thirty 336MHz Ultra Sparc 2 processors with shared memory, and its operating system is Solaris 2.6. Fortran and Java compilers are Sun Fortran F90 2.0 and Sun JDK 1.3 respectively, while the message passing library is Sun MPI 2.0. The Linux cluster consists of 16 PC computers connected with 100Mb/s Fast Ethernet switches. Every computer has an Intel Pentium III 900MHz processor with 256Mbytes of memory. The operating system is Redhat Linux 6.2, and compilers include Gnu Fortran 77 and Sun JDK 1.3. The message passing library is MPICH 1.2. For parallel computing in Java on both systems it needs a Java interface-mpiJava1.2 which binds Java code to the MPI library.

Firstly, we measure the total run time of the PIC simulation codes to compare Java performance with Fortran on the Sun E6500 and PC cluster. The Fortran compiler and corresponding options on the Sun E6500 and PC cluster are ”f90 -O5 -fast -xtarget=ultra2 -xcache=16/32/1:4096/64/1” and ”f77 -O3” respectively. We run the Java bytecodes with the JIT (just-in-time) compiler and the HotSpot server VM (Virtual Machine) on both the Sun E6500 and the PC cluster. The total time reported here excludes the initialization time. The particle data are always initialized on one processor. In this way, all the particles have the same initial states regardless of the number of processors used, and therefore the calculated energy is always the same.

Both 2D and 3D PIC codes use 32768 grid points, but their particle numbers are 1,310,720 and 294,912 respectively. We run the 2D code for 325 time steps and 3D code for 425 time steps to ensure that the beam instability is fully developed. During the process of the beam instability, some processors have more particles than others due to the particle bunch up. The maximum load imbalance observed in our code is about 10% [5]. When the codes are run on parallel computers, there is little difference for the total elapsed time on different processors. Nevertheless, we choose the longest elapsed time as our measured time.

Table 1. The run time of the separate parts in the 2D PIC code for different compiler on the Sun E6500 and the PC cluster: (a) Sun E6500; (b) PC cluster.

(a)

No. of proc.	Fortran compiler		Java com piler	
	Particle time	Field time	Particle time	Field time
1	549.86	15.60	3780.68	69.06
2	272.33	7.73	1800.93	47.33
4	136.49	4.80	893.18	28.47
8	69.45	3.94	451.70	20.13
16	39.31	5.59	243.33	13.74

(b)

No. of proc.	Fortran compiler		Java c ompiler	
	Particle time	Field time	Particle time	Field time
1	488.26	15.05	1194.92	20.10
2	239.36	8.71	592.21	15.03
4	123.19	4.57	305.87	11.64
8	68.55	4.06	158.18	9.89
16	43.20	4.28	89.41	8.64

The measured total time for the 2D and 3D codes with Java and Fortran compilers are given in Figure 3. The results show that the performance of Fortran is higher than that of Java, the performance ratio of Fortran to Java is about 6 on the Sun E6500 and 2.3 on the PC cluster. The performance of the PC cluster is higher than that of Sun E6500, but as the number of processors increases, their performance approaches each other, This is due to the higher communication overhead on the PC cluster.In Fig(a), we can find the closeness of the performance of the two Fortran implementations on Sun E6500 and PC cluster, the reason is that the Fortran on Sun E6500 is superior to the Gun Fortran on PC cluster, although the processor on Sun E6500 is 336 MHz compared with the 900 MHz processor on PC cluster.

The *field time* and the *particles time* of the 2D and 3D PIC codes are presented in Table 1 and Table 2 respectively. The *field time* includes the overheads for the *field manager* and the *field solver* which solve the Poisson equation using the FFT. The *particles time* includes the overheads for *acceleration* and *deposit* which move the particles and deposit the charge to grid points. The *particles time* contributes most to the total time, usually above 80%. In the codes, the *particles time* corresponds to the run time of the methods *push* and *depost* from the *plasma* class. From these two methods we can deduce how many floating-point operations are needed to move one particle every iteration. From this, an estimation of the performance rate, i.e. the number of the floating-point operations per second can be calculated after the real time spent on these two methods in one benchmark

IEEE COMPUTER SOCIETY

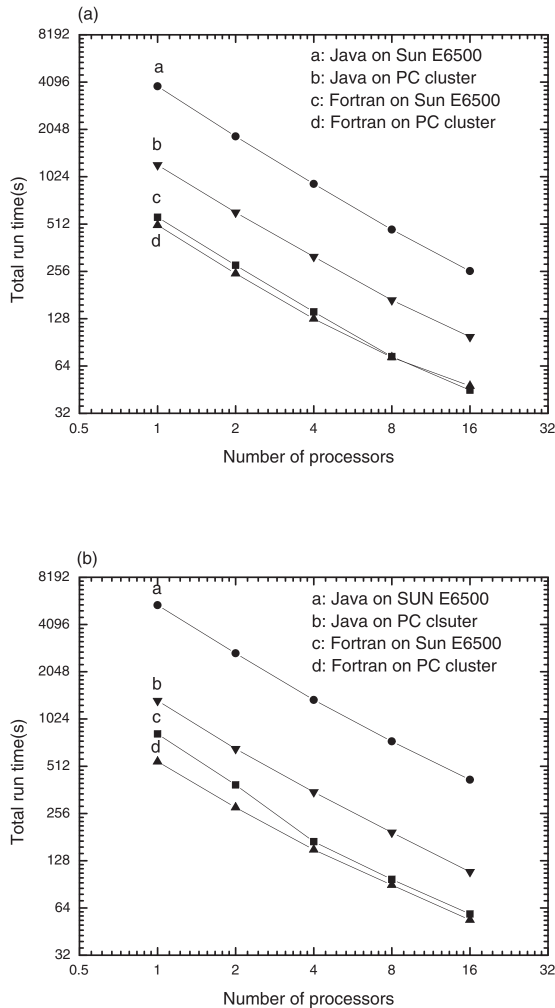

Figure 3. Total run time versus number of processors for the parallel PIC codes on the Sun E6500 and the PC cluster: (a) 2D code; (b) 3D code.

Table 2. The run time of separate parts in the 3D PIC code on the Sun E6500 and the PC cluster: (a) Sun E6500; (b) PC cluster.

(a)

No. of proc.	Fortran compiler		Java compiler	
	Particle time	Field time	Particle time	Field time
1	794.10	27.83	5222.12	126.89
2	353.82	35.06	2567.14	116.18
4	154.02	15.34	1275.15	69.92
8	83.32	14.06	663.68	73.89
16	41.02	17.53	368.08	51.67

(b)

No. of proc.	Fortran compiler		Java compiler	
	Particle time	Field time	Particle time	Field time
1	526.64	20.57	1298.93	40.23
2	264.56	17.49	627.16	33.73
4	134.77	15.28	322.09	28.48
8	74.62	14.95	168.01	25.57
16	40.32	15.36	87.40	21.38

run is known.

The 2D and 3D results are listed in Tables 3(a) and 3(b) respectively. They show that the performance rate of the 2D code is a little higher than that of the 3D code, the ratio being 1.3-1.6 on the Sun E6500 and 1.1-1.4 on the PC cluster. From the tables, we can find that the *field time* with 16 processors is longer than with 8 processors for the Fortran code, while the *field time* with 16 processors is shorter than with 8 processors for the Java code. In our implementation, the *field time* consists of two parts, which are the computation time and co-ordination overhead (including message passing, unparallelized computation and others). The computation time is found to decrease linearly with the number of processors due to low load imbalance in our code, while the co-ordination overhead becomes larger with the number of processors. When the rate of the co-ordination and computation time is large enough, it is possible that the run time becomes larger for larger number of processors. The co-ordination cost for Fortran and Java is almost the same, but the computation time for the Fortran code is shorter than for the Java code. Hence, it is easy to see that the field time is longer with 16 processors than 8 processors when using Fortran.

5 Conclusions

Today, the most widely used language for scientific computing is Fortran due to its high performance. However, Java has been getting more and more popular. It has attracted many scientists to do scientific computing in Java because it supports many new features such as portability

Table 3. The performance in Mflop/s for the PIC codes on the Sun E6500 and the PC cluster: (a) 2D code; (b) 3D code.

(a)

No. of	Sun E6500		PC cluster	
proc.	Fortran	Java	Fortran	Java
1	95.3	13.9	107.3	44.0
2	199.7	29.1	227.2	88.5
4	395.5	58.7	438.2	171.4
8	786.1	116.0	796.4	331.3
16	1527.1	215.3	1389.6	585.9

(b)

No. of	Sun E6500		PC cluster	
proc.	Fortran	Java	Fortran	Java
1	62.1	9.4	93.6	37.8
2	139.2	19.2	186.2	78.6
4	319.8	38.6	365.5	152.8
8	591.2	74.2	660.1	293.1
16	1200.8	133.8	1221.6	563.3

across platforms, powerful graphical user interface toolkits, Internet communication and protocols, etc., which are very important in modern scientific computing. The Java Grande forum was formed at the beginning of 1998 to encourage programmers to use Java as an environment for scientific computing. Members of the forum also designed a benchmark suite to evaluate Java's performance, but most of the codes are microbenchmarks or kernels.

In this paper we introduce a skeleton of a PIC code in Java. Our version was successfully implemented with the 2D and 3D plasma PIC codes. The performance of Java has been measured, and the results show that Java's performance is about 15% of Fortran on Sun E6500 and 45% of Fortran on the Linux cluster.

The performance of Java is still very low compared with other computer languages such as Fortran. Much effort needs to be invested before Java can be used as a real tool for scientific computing. However, considering that Java is a very new computer language and still under development, this looks feasible. Very promising approaches exist with the development of optimization technologies such as advanced JIT compilers [7], native code compilers which produce machine specific executables from Java source code [15], and mixed language techniques [13] which replace some time-consuming Java methods with other high performance computer languages through JNI. All these new technologies are subject to future work with our skeleton PIC simulations code. Nevertheless, even now we have enough reasons to feel optimistic about Java's future role in scientific computing.

6 Acknowledgement

The authors would like to thank the University of Wales - Cardiff for the use of their computer system Sun E6500. Special thanks go to Bryan Carpenter at Florida State University for his help in the installation of mpiJava1.2 on Sun E6500. This work was financed partly by National Science Foundation of China (NSFC) under Grants No. 40084001 and 40174041 and by the NFF initiative of the HEFCE in the U. K.

References

[1] E. Akarsu, K. Kincer, T. Haupt, and G. C. Fox. Particle-in-cell simulation codes in high performance Fortran. In *Proceedings of Supercomputing'96(IEEE, 1996)*.

[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*, San Juan, Puerto Rico, April 1999.

[3] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawak, and C. V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings of International Conference on Parallel Processing*, 1995.

[4] C. K. Birdsall and A. B. Langdon. *Plasma physics via computer simulation*. Hilger, New York, 1991.

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

[6] G. Fox and W. Furmanski. Computing on the Web - new approaches to parallel processing - Petaop and Exaop performance in the year 2007. Technical Report SCCS-784, Northeast Parallel Architectures Center, Syracuse University, 1997.

[7] T. R. Halfhill. Heating up Java. *IBM Research magazine*, 36(4), 1998.

[8] Java Benchmarks: VolancoMark. http://www.volano.com/benchmarks.html.

[9] Java Grande Benchmarks at EPCC. http://www.epcc.ed.ac.uk/javagrande/.

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

[11] G. Judd, M. Clement, and Q. Snell. DOGMA: Distributed object group metacomputing architecture. *Concurrency: Practice and Experience*, 10(11-13):977–983, 1998.

[12] Q. M. Lu and D. S. Cai. Implementation of parallel plasma particle-in-cell codes on PC cluster. *Comput. Phys. Commun.*, 135(2001):93–104.

[13] Q. M. Lu and V. S. Getov. Mixed-language high-performance computing for plasma simulation. *Scientific Programming*, 11(1):(to appear), 2003.

[14] C. Norton, B. Szymanski, and V. Decyk. Object oriented parallel computation for plasma simulation. *Communications of the ACM*, 38(10):88–100, October 1995.

[15] V. Seshadri. IBM high-performance compiler for Java. *AIXpert Mag.*, September 1997.

[16] D. W. Walker. Characterizing the parallel performance of a scale, particle-in-cell simulation code. *Concurrency: Practice and experience*, 2(4):257–288, December 1990.