

# Parallel PIC Code using Java on PC Cluster

DongSheng Cai and Quanming Lu

Institute of Information Sciences and Electronics, University of Tsukuba,  
Tsukuba 305-8573, Japan  
{qmlu,cai}@is.tsukuba.ac.jp

## Abstract

*The Java language has emerged as a dominant language that could eventually replace C++, due to its features of object-oriented, architecture neutral, multi-threaded etc. and its support for applet. But Java is believed to be "too slow" for scientific computing. Recently many high-performance PCs as those based on Pentium II have been introduced, and those PCs have a high potential for high-performance computing. We are currently building a dual PentiumPro PC cluster. In this report, using a test skeleton-PIC-code developed by Prof. V. K. Decyk of UCLA for benchmarking purpose, we have measured the performance of the Java language in serial and parallel on our PC cluster compared with the Fortran language which is the main language for scientific computing. In our benchmarking, we use JavaMPI as an interface to MPI for message passing between PCs through 100Base-TX/10Base-T ethernet switch. The benchmark results indicate that the Java language is a good candidate for scientific computing.*

**Keywords:** Java, PC cluster, Linux, PIC code, benchmark.

## 1. Introduction

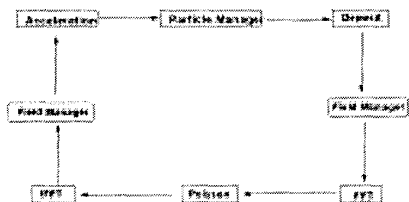
A few years ago, high performance computing is focused on supercomputers, but recently with the development of computer technology, the assumption that high performance computing will be done primarily on specialized supercomputers is questioned increasingly, the rapid progresses in performance and connectivity of ordinary workstations and PCs make it look equally possible that future of the parallel computing will be on local area networks(LAN), or even the Internet [1, 6, 7].

Now with the explosive growth of Internet came the development of a language capable of enabling cross platforms web programming: Java. The Java language has become very successful since its formal introduc-

tion in 1995. The Java source code is first compiled into platform independent bytecode, then it is interpreted by a Java Virtual Machine (JVM), so the same bytecode can be run on any platform with JVM, this characteristic of portability across platforms makes Java a natural language for network-based computing [9]. The Java is also a object-oriented language like C++, actually it is a descendant of C++, but omit various features of C++ that are considered "difficult" and charged for poor performance. And in Java language, independent threads may be scheduled on different processors by a suitable runtime, it is a multi-threading language. All of these make Java a good candidate for high performance computing [2]. Besides these, the Java's support for interactive pages to the World Wide Web, and various features such as Internet communication and protocol, graphical components, Graphical User Interface design facilities, customizable security restrictions [3]... all of these make the Java language widely accepted.

Because the Java language is designed for Internet and its characteristic of portability across various platforms, and future's parallel computing is prone to be done on local area network and Internet, it is natural to ask if the Java language is suitable for high performance computing [5, 8]. And the attractiveness of Java for scientific computing is being encouraged by bodies like Java Grande [12]. The Java Grande forum has been set up to co-ordinate the communities efforts to standardize many aspects of Java and so ensure that its future development more appropriate for scientific programmers. The goal of this report is to compare the performance of Java language with Fortran language on our PC cluster using a skeleton PIC code and then test if Java language is suitable for scientific computing. The PC cluster is named Tsukuba dual PentiumPro PC cluster, it consists of 16 HP Vectra 6/200 that is inexpensive PCs based on dual PentiumPro 200, and the PCs are interconnected through a cheap 100Base-TX/10Base-T ethernet switch, the de-

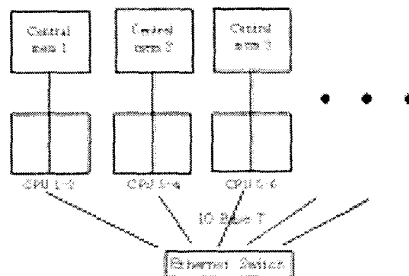
tails are described in Section 2. The skeleton PIC code which was originally written in Fortran language was proposed by Decyk as a testbed where new algorithms can be developed and tested and new computer architecture can be evaluated [4, 13]. It has been deliberately kept minimum, but includes all the essential pieces for depositing charge, advancing particles, and solving the field. It moves only electrons, with periodic electrostatic forces obtained by solving Poisson's equation with the fast Fourier transforms, and uses the electrostatic approximation and magnetic fields are neglected. The only diagnostic is particle and field energy. The basic structure of the main loop of the skeleton code is illustrated in Figure 1. The benchmark results are described in Section 3.



## 2. Tsukuba dual PentiumPro PC Cluster

We have built a PentiumPro PC cluster which consists of 16 PCs, and the operating system we used is Redhat Linux 5.2. Each PC is HP Vectra 6/200, and is a 200 MHz dual PentiumPro SMP machine with 128 Mbyte EDD DIMM memories (Fig.2). The Java language we used is JDK1.1.7 with just-in-time (JIT) compiler, the JIT compiler translates Java bytecode into native code at run time. The Fortran compiler we choose is Gnu f77, both Java and Fortran compiler are free software, this is also the reason why we choose Gnu f77 as Fortran compiler. On parallel computation, for Fortran we use MPI as message passing library [15], for Java the message passing library is also MPI, but with an interface—JavaMPI which uses a Java-to-C interface generator to define a Java wrappers from the C MPI header [14].

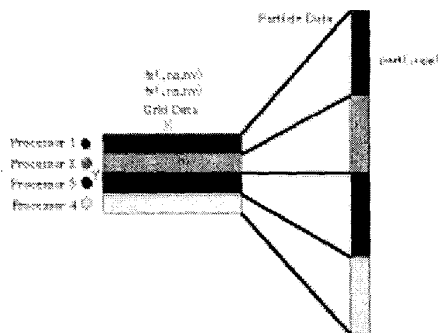
All the benchmarks of the code are run in double precision. The skeleton PIC code is developed by V.



Decyk of UCLA. The code is kept as small as possible. However, it contains all essences of PIC code as shown in Fig.1. The physical problem in the code is a beam plasma instability where 10% of the particles are the beam whose velocity is five times the thermal velocity of the background electrons, and the quadratic spline function is used for the interpolation between grids and particles. In the parallel benchmarks, the benchmark time excludes the initialization and always initialized on one processor, then the initialized particle data are distributed to other processors. This is because the initialization time is negligible and we want to start the simulation always in the same state and the same total energy even using parallel computers with different number of processors. In our skeleton PIC code, a one-dimensional partition as show in Fig. 3 is used, different processors are assigned to different spatial regions and particles are assigned to processors according to the spatial region they are belong to. As particles move from one region to another, they are assigned to the processor which is associated with the new region.

## 3. Benchmark Results

Now the most wide-accepted scientific computing language is Fortran. We first benchmark the performance of the 2D skeleton PIC code using the Java language on single PentiumPro processor, comparing with the Fortran language. The Java language is an object-oriented language, our code consists of two classes: one is named "plasma" (there is two methods: "push" is to move the particles, and "deposit" is to deposit the charge), the other is "field" (there is also two methods: "pois" is to solve the poission equation, and "fft" is to implement the FFT transform). The main procedure of the code is described as following:



```

plasma plasma =new plasma();
field filed=new new filed();

for(int k=0; k<nloop; k++){

    .....

    isign=-1;
    field.fft(isign,qc);
    field.pois(isign,qc,fxc,fyc);

    isign=1;
    field.fft(isign,fxc);
    field.fft(isign,fyc);

    .....

    plasma.push(fx,fy,dt);
    plasma.depost(q);

}

```

For this benchmark purpose, the problem uses 8192 grids and 327680 particles, and the time steps are 325 so that the beam instability is fully developed. In the skeleton PIC code most of the CPU time is spent on both the particle acceleration that is the method "push" of class "plasma" and the deposition that is the method "depost" of class "plasma". From these two methods we can know how many floating operations needed to move one particle in one iteration. Then the actual performance, i.e. the number of the floating operations per second can be calculated after the real time spent on these two methods in one benchmark run is known. The measured benchmark results are listed in Table 1. The results indicate that the performance of Java on Linux operating system

is about 12 and 18 times slower than that of Gnu f77 and PGF77, and on Windows NT 4.0 operating system the performance of Java is about 3 time slower than that of Visual Fortran 5.0, PGF77 and Visual Fortran 5.0 are commercial software which are developed by Portland Group Inc. and Microsoft corporation respectively. If we run the Java bytecode with HotSpot [11], the performance of Java can be improved about 35%, and attained about 37% of Visual Fortran 5.0. The Java HotSpot<sup>TM</sup> performance engine is an add-on performance module for the Java<sup>TM</sup> 2 SDK. The Java HotSpot performance engine employs state-of-the-art technology to offer many performance enhancements: adaptive compiler, improved garbage collection, thread synchronization. Its main function is to detect the performance bottlenecks(which is also called "hot spots") of the code, then do optimization on that part to improve performance.

#### Single Processor Benchmark

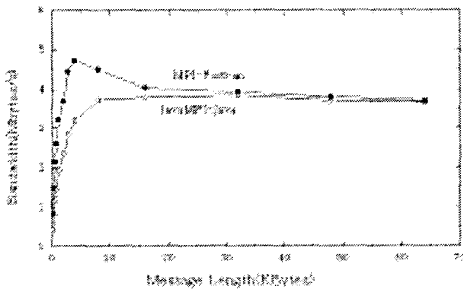
Operating System	Compiler & Option	benchmark results (Mflops) (% peak)
Redhat 5.2	Gnu f77 -O3	26.2(13.1%)
	pgf77 -O2 -Munroll -tp p6 -pc 64	38.2(19.1%)
NT 4.0	JDK1.1.7	2.2(1.1%)
	Visual Fortran 5.0	32.4(16.2%)
	JDK1.2.2(without HotSpot 1.01 beta)	8.0(4.0%)
	JDK1.2.2(with HotSpot 1.01 beta)	12.1(6.1%)

Table 1: Benchmark results with the 2D skeleton PIC code using different compilers on different operating systems. In these benchmarks, we use 8192 grids and 327680 particles, the time steps are 325.

We also benchmark the performance of the 2D skeleton PIC code on our PC cluster using Java and Fortran language. For Fortran language the message passing library we used is MPI provided by University of Notre Dame, and for Java language we use the JavaMPI software developed in University of Westminster. In JavaMPI, a Java-to-C interface generator(JCI) is used to provide a native interface for the MPI library and generate files that enable MPI calls from Java. Two Java classes are generated by this software: **MPI** and **MPIconst**: The **MPI** class contains all of the methods required to call native MPI functions. The **MPIconst** class encapsulates all of the MPI constants.

First we compare the message passing performance of MPI and JavaMPI. In this program an increasing sized message is sent back and forth between two pro-

processors, from the message passing time we can measure the bandwidth for both MPI and JavaMPI, the benchmark is based on standard blocking **Send** and **Recv** which we use through our report. The results are shown in Fig. 4, when the message size is very small, the message passing performance for MPI is higher than for JavaMPI, but when the message size is large, there is no distinct difference, and the peak bandwidth occurs at about 4 Kbytes. In our benchmark code, the message length is from 1 Kbytes to 15 Kbytes, so the JavaMPI will not degrade the message passing performance compared with MPI.



Benchmark on PC cluster using Java

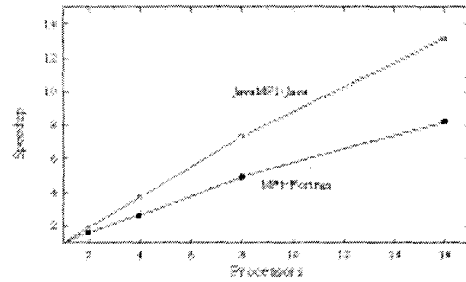
Processor number	total time(s)	speedup	efficiency(%)
1	6288	1.00	-
2	3320	1.89	95
4	1698	3.70	93
8	858	7.33	92
16	479	13.13	82

Table 2a: Benchmark results of our PentiumPro PC cluster for 2D skeleton PIC code using Java language. In these benchmarks, we use 8192 grids and 327690 particles, and the time steps are 325.

Benchmark on PC cluster using Fortran

Processor number	total time(s)	speedup	efficiency(%)
1	517	1.00	-
2	320	1.62	81
4	197	2.62	66
8	105	4.92	62
16	63	8.21	51

Table 2b: Benchmark results of our PentiumPro PC cluster for 2D skeleton PIC code using Fortran language. In these benchmarks, we use 8192 grids and 327690 particles, and the time steps are 325. The compiler options are "f77 -O3"



Then we measure the speedup and efficiency for Java and Fortran. In this benchmark, we use 8192 grids and 327680 particles for 325 time steps. The results are described in Table 2 and Fig.5, the performance of Java is about 10 time slower than that of Fortran. But the efficiency for Java is obviously higher than for Fortran, and it get an almost linear speedup. From the definition of efficiency  $E$ , we can easily infer this results:

$$E = \frac{T_1}{p(T_p + T_C)}$$

where  $p$  is the number of processors,  $T_1$  and  $T_p$  is the computational time for one processor and  $p$  processors respectively, and  $T_C$  is the overhead mainly caused by message passing.

In our code, we can treat  $T_1 \approx pT_p$  due to the low load imbalance which is typically 10%, then

$$E = \frac{1}{1 + \frac{T_C}{T_p}}$$

As we have mentioned before, there is no distinct difference of message passing performance between MPI

and JavaMPI in our benchmark, but the computational time of Java is much larger than that of Fortran. So efficiency of Java is higher.

In our code the message passing time consists of two main parts:  $T_{C1}$  where the particles move from one processor to other processors,  $T_{C2}$  where the fields are transposed in FFT. If we use same grid number, keeping the number of particles constant and only changing the number of processors has no distinct influence on  $T_{C1}$  and  $T_{C2}$ , the efficiency will degrade with the increasing of the number of processors; while keeping the number of processors constant, increasing the number of particles will increase  $T_{C1}$  but has no effect on  $T_{C2}$ , so the efficiency will increase. We measure efficiency as a function of particle number and processor number using JavaMPI on our Pentiumpro cluster. In the benchmark problem, we use 8192 grids and run the problem 325 time step. The results are shown in Table 3. increasing the number of processors will degrades efficiency and increasing the number of particles will increase efficiency, and we can increase the number of particles almost linearly with respect to number of processors  $p$  to maintain same efficiency, the system is highly scalable.

Benchmark on PC cluster using Java

n	p=1	p=2	p=4	p=8	p=16
9	100	92	90	89	73
18	100	94	92	91	82
36	100	95	94	92	83
72	100	97	96	95	90
144	100	99	98	97	94
288	100	100	99	99	96

Table 3: Efficiency as a function of  $n$  and  $p$  for 2D skeleton-PIC code on our PC cluster,  $n$  is the average particle number on one grid,  $p$  is the processor number. We use 8192 grids and 325 time steps.

#### 4. Conclusion

Due to Java's many superiorities over C, C++ and Fortran, Java is getting more and more popular. Java is already being adopted in many entry level college programming courses and will surely be attractive for teaching in middle or high schools. But there is one basic reason that Java is often claimed to be inappropriate for scientific computing: Java is generally interpreted and is therefore unable to run at machine speed.

Using a two-dimensional skeleton PIC code, we benchmark the performance of Java on our PC cluster, and attain very high efficiency about 82% for 16

processors, which indicates that Java language is suitable for parallel computing on PC cluster, although its performance is very slow compared with Fortran on Linux operating system. On Windows NT 4.0 operating system, Java's performance is also very high, it can attain about half of that of Fortran when we use HotSpot. Considering many advantages of Java over Fortran, it is worth doing scientific computing using Java. Java is a very new computer language and still under development, with the development of JIT, HotSpot, native code compilers which produce machine specific executables from Java source code and other optimization compilers for Java language on various operating system [10], we can hope the Java language can attain comparable performance of Fortran which is the main language of scientific computing.

#### References

- [1] D. S. Cai, Q. M. Lu, and Y. T. Li. Scalability in particle-in-cell code using both PVM and OpenMP on PC cluster. In *Proceedings of 3rd Workshop on Advanced Parallel Processing Technologies*, pages 69-73, Changsha, China, October 1999.
- [2] B. Carpenter, Y. J. Chang, G. Fox, D. Leskiw, and X. M. Li. Experiments with HPJava. *Concurrency: Practice and Experience*, 9(6):633-648, June 1997.
- [3] G. Cornell and C. S. Horstmann. *CoreJava*. Sunsoft Press, Mountain View, CA, 1996.
- [4] V. K. Decyk. Skeleton PIC codes for parallel computers. *Computer Physics Communications*, 87:87-94, 1995.
- [5] G. Fox. Java for parallel computing and as a general language for scientific and engineering simulation and modelling. *Concurrency: Practice and Experience*, 9(6):415-425, June 1997.
- [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] A. Geist, A. Beguelin, J. Dongarra, W. Jinag, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine - A users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, 1994.
- [8] V. Getov, S. Flynn-Hummel, and S. Mintchev. High-performance parallel programming in Java: Exploiting native libraries. In *Proceedings of the 1998 ACM Workshop on Java for High-Performance Network Computing*, Palo Alto, California, 1998.
- [9] M. Gudd, M. Clement, and Q. Snell. Dogma: Distributed object group management architecture. Technical Report TR BYU-NCL-97-102, Computer Science Department, Brigham Young University, 1997.
- [10] T. R. Halfhill. Heating up Java. *IBM Research magazine*, 36(4), 1998.

- [11] HotSpot. <http://developer.java.sun.com/developer/earlyaccess/hotspot/index.html>.
- [12] Java Grande Forum. <http://www.javagrande.org/>.
- [13] P. C. Liewer and V. K. Decyk. A general concurrent algorithm for plasma particle-in-cell simulation codes. *J. Computational Physics*, 85(2):302–322, December 1989.
- [14] S. Mintchev and V. Getov. Towards portable message passing in Java: binding MPI. In *Proceedings of EuroPVM-MPI*, pages 135–142, Kraków, Poland, November 1997. Springer LNCS 1332.
- [15] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1996.