Computer Physics Communications 135 (2001) 93–104

Computer Physics Communications

www.elsevier.nl/locate/cpc

Implementation of parallel plasma particle-in-cell codes on PC cluster

Quan Ming Lu [a,*], Dong Sheng Cai [b]

[a] Department of Earth and Space Sciences, University of Science and Technology of China, Hefei, Anhui 230026, People's Republic of China
[b] Institute of Information Sciences and Electronics, University of Tsukuba, Tsukuba, Ibaraki 305, Japan

Received 5 June 2000; received in revised form 26 October 2000; accepted 12 November 2000

Abstract

Plasma particle-in-cell (PIC) codes model the interaction of charged particles with the surrounding fields, and they have been implemented on many advanced parallel computers. Recently, many PC clusters which consist of inexpensive PCs have been developed to do parallel computing, and we also build such a PC cluster. In this paper, we present the implementation of a parallel plasma PIC code on our PC cluster using MPI, PGHPF and JavaMPI. © 2001 Elsevier Science B.V. All rights reserved.

PACS: 07.05.Tp

Keywords: PIC code; PC cluster; MPI; PGHPF; Java

1. Introduction

A plasma particle-in-cell (PIC) code follows the orbits of particles in the surrounding fields which are calculated self-consistently from the charge and/or current densities created by these same particles [1,2]. Each time step in a PIC code consists of two major steps: the *particle push* to update the particle orbits and calculate the new charge and/or current densities, and the *field solver* to update the surrounding fields. Since the particles can be located anywhere within the simulation domain but the surrounding fields are defined only on discrete grid points, the *particle push* uses two interpolation steps to links the particle orbits and the fields: a "gather" step to interpolate fields from the grid points to particle positions and a "scatter" step to deposit the charge and/or current of each particle to grid points. PIC particle simulation has long been used by scientists to study the nonlinear kinetic problems in space and laboratory plasma physics. But only after the emergence of parallel computers, does PIC particle simulation with large number of particles become possible. Actually, parallel PIC particle codes have been implemented on parallel supercomputers using the General Concurrent PIC (GCPIC) algorithm which uses a domain decomposition to distribute the computation among the processors [3–5] and other decomposition methods [6–8].

[*] Corresponding author.
E-mail address: qmlu@yahoo.com (Q.M. Lu).

But recently with the rapid progress in performance and connectivity, networks of PCs and workstations are becoming an appealing vehicle for parallel computing. By relying solely on commodity hardware and software, networks of PCs and workstations can offer parallel processing at low cost. Hence, parallel computing on commodity hardware is gaining wide acceptance over traditional high cost supercomputers. This new wave in parallel computing is popularly called cluster computing. In cluster based parallel computing, PCs or workstations are interconnected together using a low-latency and high bandwidth interconnection networks [9–11]. Using the same technologies, we have built such a PC cluster, it consists of 10 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 details of the PC cluster are described in Section 2. And in Section 4, we present the implementation of a parallel plasma PIC code using different parallel programming models and languages on our PC cluster. The details of the parallel PIC code are described in Section 3.

2. Dual PentiumPro PC cluster

We have built a PentiumPro PC cluster, it consists of 10 dual processors PC computers. Each PC is a HP Vectra 6/200 machine with dual PentiumPro 200 MHz processors and 128 Mbyte EDD DIMM memories. As shown in Fig. 1, the PCs are interconnected with a 100Base-TX/10Base-T ethernet switch. The operating system we used is Redhat Linux 5.2, and we upgrade the kernel to version 2.2.10. In order to do parallel computing, we must install compilers and message passing libraries on our PC cluster. The compilers include free software (Gun f77 and JDK1.1.7 [12]), and commercial software (PGF77 and PGHPF which are from Portland Group Inc.). The message-passing libraries include LAM-MPI 6.2 provided by University of Notre Dame [13], and JavaMPI which uses a Java-to-C interface generator (JCI) to define a Java wrappers from C MPI header, JavaMPI is developed by University of Westminister and used to do parallel computing with Java language [14].

3. Parallel PIC code

The parallel PIC code we used is a skeleton PIC code which was proposed by Decyk as a testbed where new algorithms can be developed and new computer architecture can be evaluated, it was originally written using Fortran language with message-passing library [15]. The code is an electrostatic code and magnetic fields are neglected,

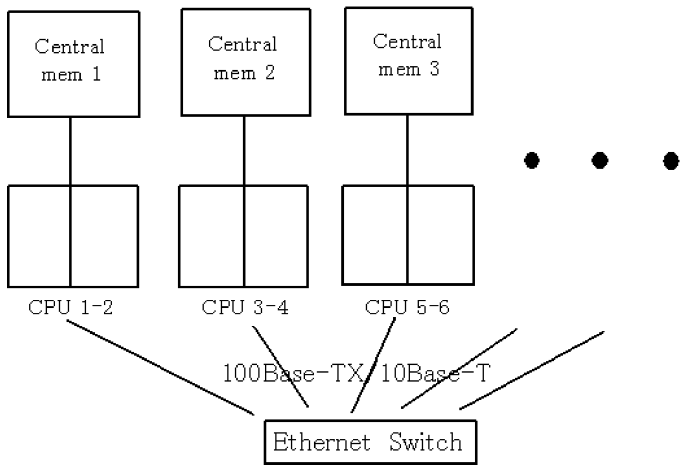


Fig. 1. Distributed shared memory PC cluster (Dual-PentiumPro cluster).

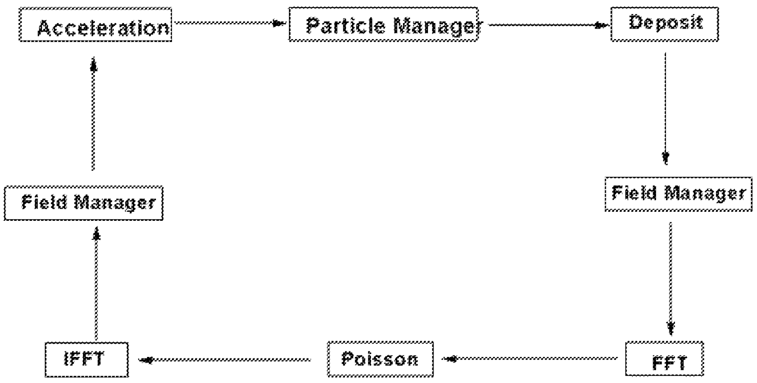


Fig. 2. Structure of the main loop of skeleton PIC codes.

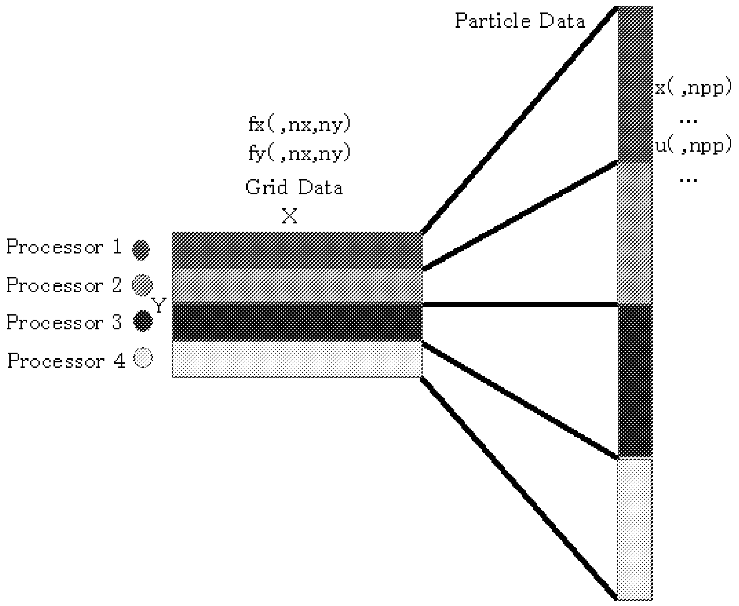


Fig. 3. Grid and particle partition.

and it only moves electrons. Although this code has been deliberately kept minimum, it includes all the essential pieces which are depositing charge, advancing particles and solving fields. And the periodical boundary condition is used when solving Possion's equation with the fast Fourier transforms. The only diagnostic in this code is particle and field energy. The basic structure of the main loop of the skeleton code is illustrated in Fig. 2. The physical problem in the code is a beam plasma instability where 10% of the particles are the beam whose beam velocity is five times the thermal velocity of the background electrons. In the code, the quadratic spline function is used for the interpolation between grids and particles, and all the variables are in double precision. In the parallel-benchmarks, the benchmark time excludes the initialization and the particle data are 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 the skeleton PIC code, we use GCPIC algorithm to distribute the particle and field data, 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 regions 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. The maximum load imbalance observed in this code is about 10%.

4. Implementation of parallel PIC code

Usually, there are two types of parallel programming models on distributed system including PC cluster. First is the message-passing programming model, in this model the calculations are distributed into processors, and each processor has its own data. All the processors are run concurrently, and each processors must use explicit send/receive directives to send/get the data from other processors. The most popular message passing library is MPI and PVM. The second is data parallel programming model, in this model calculations are performed concurrently with data distributed across processors and each processor processes its own segment of data. The most famous data parallel compiler is HPF [16]. In this section we use these two parallel programming models to implement the parallel PIC code on our PC cluster. We also implement this code using Java language with the message-passing programming model, the message passing library is MPI, but with a JavaMPI to define an interface to it.

4.1. Message-passing programming model

We use the MPI message passing library to realize the message-passing parallel programming model, the details can be found in [15]. In this model, we must use a Fortran compiler to generate a executable file while linking the MPI message passing library, here we use PGF77 compiler. First of all we measure the speedup and efficiency of the 2D and 3D parallel PIC code. The results are listed in Table 1. The speedup S and efficiency E are defined as following:

Table 1
(a) Benchmark results of the 2D parallel skeleton PIC code on our PC cluster. We use 32 768 grids and 1 310 720 particles, the time steps are 325. The compiler options are "pgf77 -O2 -Munroll -tp p6 -pc 64". (b) Benchmark results of the 3D parallel skeleton PIC code on our PC cluster. We use 32 768 grids and 995 328 particles, the time steps are 425. The compiler options are "pgf77 -O2 -Munroll -tp p6 -pc 64"

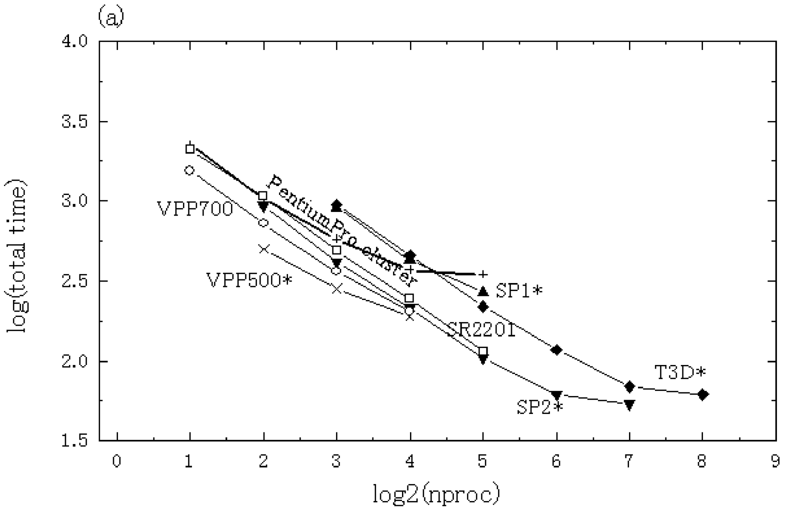
Processor number	Total time (s)	Speedup	Efficiency (%)
(a)			
1	1731	1.00	–
2	866	2.00	100
4	435	3.98	99
8	247	7.00	88
16	173	10.00	63
(b)			
1	5610	1.00	–
2	3044	1.84	92
4	1514	3.70	93
8	978	5.74	72
16	716	7.84	49

$$S = \frac{T_1}{T_p}, \qquad E = \frac{T_1}{pT_p},$$

where p is the number of processors, T_1 and T_p is the total time for one processor and p processors, respectively.

In the benchmark problem, for 2D case we use 32 768 grids and 1 310 720 particles, and get a 63% efficiency for 16 processors; For 3D case we use 32 768 grids and 995 328 particles, and get a 49% efficiency for 16 processors.

Next we compare the performance of our PC cluster with some supercomputers. We run the 2D parallel PIC code for 325 time steps with 32 768 grids and 3 571 712 particles, and the 3D parallel PIC code for 425 time steps with 262 144 grids and 7 962 624 particles. The results are shown in Figs. 4(a) and (b). As indicated in the figures, the performance of the PC cluster is comparable with some new supercomputers, but cost of our PC cluster is only 1–10% of these supercomputers.



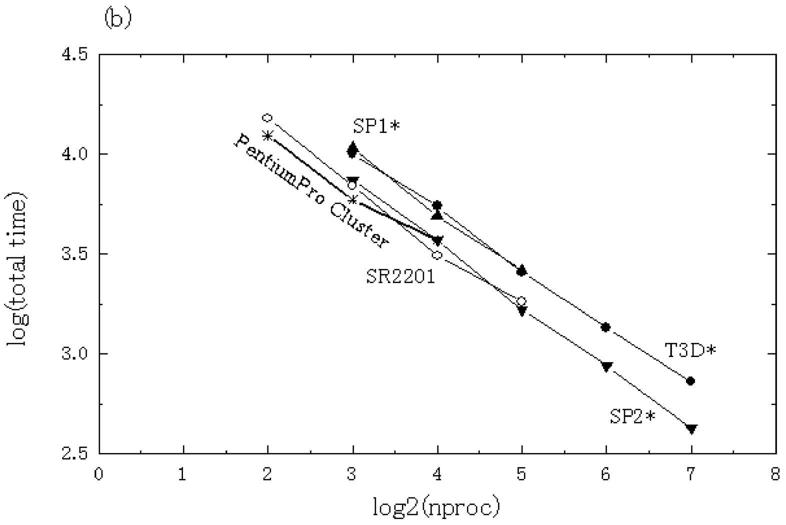


Fig. 4. Total time versus number of processors on log-log scale for various parallel computers, the compiler options for our PentiumPro cluster is "pgf77 -O2 -Munroll -tp p6 -pc 64". (a) 2D code: 32 768 grids and 3 571 712 particles for 325 time steps. (b) 3D code: 262 144 grids and 7 962 624 particles for 425 time steps. The superscript * means that the data are provided by Professor V.K. Decyk of UCLA.

4.2. Data parallel programming model

HPF provides a data parallel model of computations. In this model calculation are performed concurrently over data distributed across processors. Each processor processes the data of its own and the HPF compiler can detect concurrent calculations with distributed data. The sections of distributed data can be processed in parallel if there are no dependencies between them. HPF has a limitation in expressing pipelined computations essential for parallel processing of distributed data with dependencies between sections, HPF also has some other disadvantages: overhead introduced by the compiler, unknown performance of operations with distributed arrays, and additional memory required for storing array with an alternative distribution [17], the significant advantage of using HPF is that the conversion from f77 to HPF results in a well structured easily maintained portable program, and a HPF code can be developed on one machine and run on another. The 1D and 2D electrostatic PIC codes have been implemented in HPF on an IBM SP-2 [18], a near linear speed-up was obtained and its performance was comparable to the message-passing implementations. In our paper, we implement the 2D and 3D electrostatic codes on our inexpensive PC cluster, the data parallel compiler we used is PGHPF 2.4 which was developed by the Portland Group Inc. In our code, we first use ALIGN directive to coalign the same group of arrays. Then each group of coaligned arrays is distributed on to abstract processors with DISTRIBUTE directives, and use INDEPENDENT directive and HPF library intrinsics to ensure operations are performed concurrently on segments of data owned by each processor. At the same time, the data communication between processors are performed by HPF library intrinsics. Followings we give the example of "push" subroutine performed using PGHPF.

PGHPF

```
subroutine push:
dimension x(npmax,nvp),y(npmax,nvp),
u(npmax,nvp),v(npmax,nvp),                             C nvp is the number
sbufl(idimp,nbmax,nvp),sbufr(idimp,nbmax,nvp)            of processors
!HPF$  ALIGN (*,:) with x(*,:)::y,u,v
!HPF$  ALIGN (*,*,:) with sbufl(*,*,:)::sbufr
!HPF$  PROCESSORS pr(nvp)
!HPF$  DISTRIBUTE x(*,BLOCK) onto pr
!HPF$  DISTRIBUTE sbufl(*,*,BLOCK) onto pr
!HPF$ INDEPENDENT
do 100 i=1,nvp
do 200 j=1,npp(i)
....                                                   C here move the particles
define particles which move to other processors
200  continue
100  continue
end

subroutine pmove                                   C this subroutine move
                                                     particle to other processors
...
end
```

We run the 2D parallel PIC code with 32 748 grids and 1 310 720 particles and 3D parallel PIC code with 32 748 grids and 995 328 particles, the compiler and options for single processor is "pgf77 -O2 -Munroll -tp p6 -pc 64". The results are shown in Table 2. Compared with the performance of MPI, the performance of PGHPF is a little lower than that of MPI, which is about 15%.

Table 2
(a) Benchmark results of PGHPF on our PC cluster for the 2D parallel skeleton PIC code. In these benchmarks, we use 32 768 grids and 1 310 720 particles, and 325 time steps. The compiler options for PGHPF are "pghpf -O2 -Munroll -tp p6 -pc 64". (b) Benchmark results of PGHPF on our PC cluster for the 3D parallel skeleton PIC code. In these benchmarks, we use 32 768 grids and 995 328 particles, and 425 time steps

Processor number	Total time (s)	Speedup	Efficiency (%)
(a)			
1	1731	1.00	–
2	902	1.92	96
4	477	3.63	91
8	293	5.91	74
16	243	7.12	45
(b)			
1	5610	1.00	–
2	3152	1.78	89
4	1589	3.53	88
8	1091	5.14	64
16	863	6.50	41

4.3. *Implementation of PIC code using JavaMPI*

The Java language has become very successful since its formal introduction in 1995. The Java source code is first compiled into platform independent bytecode, then it is interpreted by a Java Virtual Machine (JVM), therefore the same bytecode can be run on any platform with JVM, this characteristic of portability across platforms makes

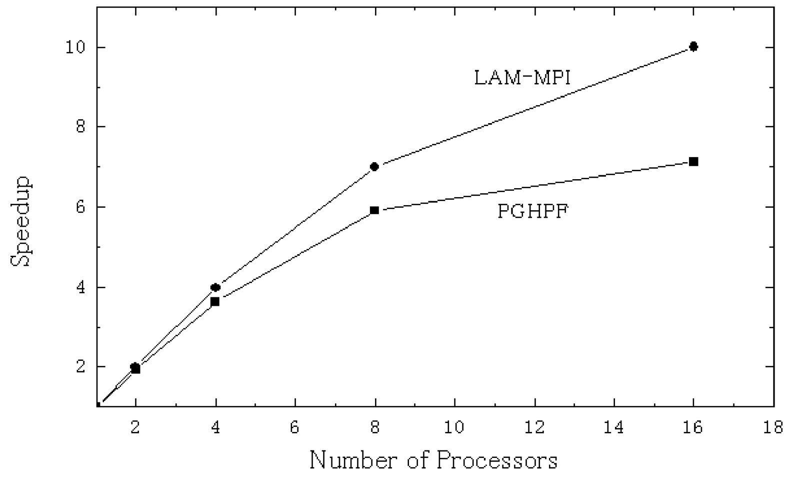


Fig. 5. Speedup of the 2D parallel skeleton PIC code using PGHPF and MPI for different processors. The time steps are 325, the grids and particles are 32 768 and 1 310 720, respectively. The compiler options for PGHPF and MPI are "pghpf -O2 -Munroll -tp p6 -pc 64" and "pgf77 -O2 -Munroll -tp p6 -pc 64".

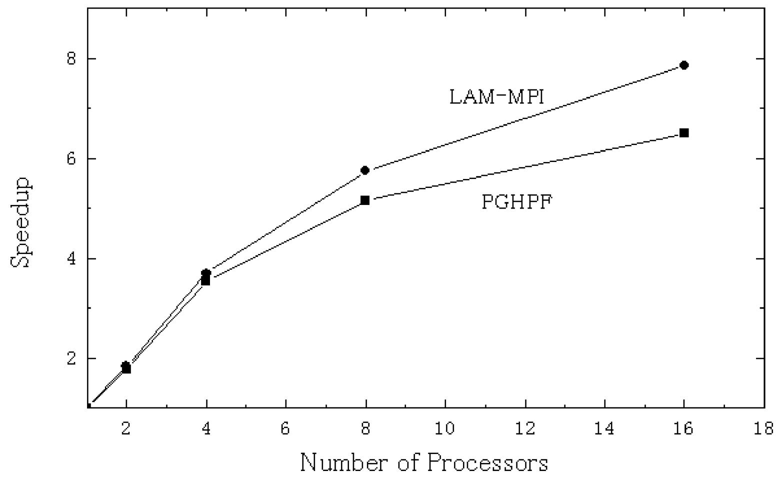


Fig. 6. Speedup of the 3D skeleton parallel PIC code using PGHPF and MPI for different processors. The time steps are 425, the grids and particles are 32 768 and 995 328, respectively. The compiler options for PGHPF and MPI are "pghpf -O2 -Munroll -tp p6 -pc 64" and "pgf77 -O2 -Munroll -tp p6 -pc 64".

Java a natural language for network-based computing [19]. The Java is also a object-oriented language like C++, actually it is a descendant of C++, but omits various features of C++ that are considered "difficult" and charged for poor performance. And in the 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 [20]. 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 [11]... all of these make the Java language widely accepted and suitable for high performance computing. And the attractiveness of Java for scientific computing is being encouraged by bodies like Java Grande [21]. 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. We also implement the parallel PIC code using Java to test if Java language is suitable for scientific computing.

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 are two methods: "push" is to move the particles, and "depost" is to deposit the charge), the other is "field" (there are also two methods: "pois" is to solve the possion 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 327 680 particles, and the time steps are 325. 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 3. 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, Visual Fortran 5.0 is commercial software which is developed by Microsoft corporation. If we run the Java bytecode with HotSpot [22], the performance of Java can be improved about 35%, and attained about 37% of Visual Fortran 5.0. The Java HotSpot^{TM} performance engine is an add-on performance module for the Java^{TM} 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.

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

Table 3
Benchmark results with the 2D skeleton PIC code using different compilers on different operating systems. In these benchmarks, we use 8192 grids and 327 680 particles, the time steps are 325

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%)
	JDK1.1.7	2.2 (1.1%)
NT 4.0	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%)

Dame, and for Java language we use the JavaMPI software developed in University of Westminister. In JavaMPI, a Java-to-C interface generator (JCI) is used to provide an 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.

We measure the speedup and efficiency for Java and Fortran. In this benchmark, we use 8192 grids and 327 680 particles for 325 time steps. The results are described in Table 4 and Fig. 7, the performance of Java is about 10

Table 4
(a) Benchmark results of our PentiumPro PC cluster for 2D skeleton PIC code using Java language. In these benchmarks, we use 8192 grids and 327 690 particles, and the time steps are 325. (b) Benchmark results of our PentiumPro PC cluster for 2D skeleton PIC code using Fortran language. In these benchmarks, we use 8192 grids and 327 690 particles, and the time steps are 325. The compiler options are “f77 -O3”

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
Benchmark on PC cluster using Fortran			
1	517	1.00	–
2	320	1.62	81
4	197	2.62	66
8	105	4.92	62
16	63	8.21	51

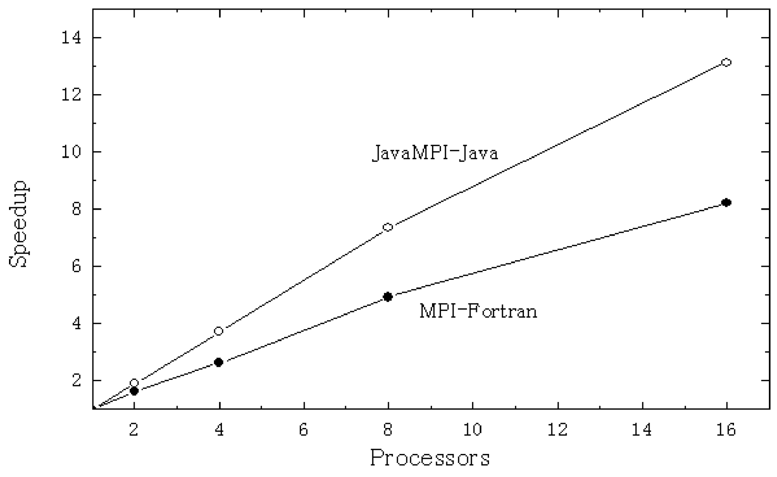


Fig. 7. Speedup versus number of processors for Java and Fortran on PC cluster, the 2D skeleton PIC code uses 8192 grids and 327 690 particles, and the time steps are 325.

time slower than that of Fortran. However, Java has a higher efficiency than Fortran, and it gets an almost linear speedup. From the definition of efficiency, we can infer this results, the total time T_p for p processors is equal computational time T_C plus overhead T_O, and T_O is mainly caused by message passing. Therefore,

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

In our code, we can treat $T_1 \approx pT_p$ due to the low load imbalance, then

$$E = \frac{1}{1 + T_O/T_C}.$$

As there is no distinct difference of message passing performance between MPI and JavaMPI in our benchmark [23], but the computational time of Java is much larger than that of Fortran, therefore efficiency of Java is higher.

5. Conclusion

We have built an inexpensive PC cluster for scientific computing, and the PC cluster consists of 10 dual PentiumPro PCs. We implement the parallel PIC code using MPI, PGHPF and JavaMPI, and the code is implemented using the General Concurrent PIC (GCPIC) algorithm which uses a domain decomposition to devide the computation among the processors. When the number of processors we used is not large, our PC cluster can get a comparable performance with some new supercomputers, but obviously with a higher performance/cost ratio due to its lower cost. The parallel data programming compiler PGHPF also can get a high performance, with a little lower performance than the message-passing programming model, both of these parallel programming models are suitable for scientific computing on PC cluster. The Java language also can be used to do parallel computing on PC cluster, and their performance is very low, considering that 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 systems [24], we can hope that Java language can attain comparable performance of Fortran and suitable for scientific computing.

Acknowledgements

One of the authors (Q.M. Lu) was supported by the National Science Foundation of China (No. 49834030, 40084001).

References

[1] C.K. Birdsall, A.B. Langdon, Plasma Physics via Computer Simulation, McGraw-Hill, New York, 1985.
[2] R.W. Hockney, J.W. Eastwood, Computer Simulation Using Particles, McGraw-Hill, New York, 1981.
[3] R.C. Liewer, V.K. Decyk, A general concurrent algorithm for plasma particle-in-cell simulation codes, J. Comput. Phys. 85 (1989) 302–322.
[4] R.C. Liewer, V.K. Decyk, J.M. Dawson, B. Lembege, Numerical studies of electron dynamics in oblique quasi-perpendicular collisionless shock waves, J. Geophys. Res. 96 (1991) 9455–9465.
[5] T.J. Kruchen, P.C. Liewer, R.D. Ferraro, V.K. Decyk, A 2D electromagnetic PIC code for distributed memory parallel computers, in: Proceedings of the 6th Distributed Memory Computing Conference, IEEE Computer Society Press, Los Alamitos, CA, 1991, p. 452.
[6] D.W. Walker, Particle-in-cell plasma simulation codes on the connection machines, Comput. Systems in Engrg. 2 (1991) 307–319.
[7] N.G. Azari, S.Y. Lee, Hybrid task partitioning for particle-in-cell simulation on shared memory systems, in: Proceedings of Internat. Conf. on Distributed Computing Systems, Ballas, TX, 1991, p. 526.

[8] S.Y. Lee, N.G. Azari, Hybrid task decomposition for particle-in-cell methods on message passing systems, in: Proceedings of International Conference on Parallel Processing, St. Charles, IL, Vol. 3, 1992, p. 141.
[9] D.J. Becker, T. Sterling, D. Savarese, J.E. Dorband, U.A. Ranawak, C.V. Packer, Beowulf: A parallel workstation for scientific computation, in: Proceedings of International Conference on Parallel Processing, 1995.
[10] C. Reschke, T. Sterling, D. Ridge, D. Savarese, D. Becker, P. Merkey, A design study of alternative network topologies for the beowulf parallel workstation, in: Proceedings of High Performance and Distributed Computing, 1996.
[11] D.S. Cai, Q.M. Lu, Y.T. Li, Scalability in particle-in-cell code using both PVM and OpenMP on PC cluster, in: Proceedings of 3rd Workshop and Advanced Parallel Processing Technologies, Changsha, China, October 1999, pp. 69–73.
[12] G. Cornell, C.S. Horstmann, CoreJava, Sunsoft Press, Mountain View, CA, 1996.
[13] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, J. Dongarra, MPI: The Complete Reference, MIT Press, Cambridge, MA, 1996.
[14] S. Mintchiev, V. Getov, Towards portable message passing in Java: binding MPI, in: Proceedings of EuroPVM-MPI, Krakow, Poland, Lecture Notes in Comput. Sci., Vol. 1332, Springer, Berlin, November 1997, pp. 135–142.
[15] V.K. Decyk, Skeleton PIC codes for parallel computers, Comput. Phys. Commun. 87 (1995) 87–94.
[16] I.T. Foster, Designing and Building Parallel Programs, Addison-Wesley, Reading, MA, 1994.
[17] M. Frumkin, H. Jin, J. Yan, Implementation of NAS parallel benchmarks in high performance Fortran, NAS Technical Report NAS 98-009, September 1998.
[18] E. Akarsu, K. Dincer, T. Haupt, G.C. Fox, Particle-ib-cell simulation codes in high performance Fortran, Supercomputing (1996).
[19] M. Gudd, M. Clement, Q. Snell, Dogma: Distributed Object Group Management Architecture, Technical Report TR BYU-NCL-97-102, Computer Science Department, Brigham Young University, 1997.
[20] B. Carpenter, Y.J. Chang, G. Fox, D. Deskiw, X.M. Li, Experiments with HPJava, Concurrency: Practice and Experience 9 (6) (1997) 633–648.
[21] Java Grande Forum: http://www.javagrange.org/.
[22] HotSpot, http://developer.java.sun.com/developer/earlyaccess/hotspot/index.html.
[23] D.S. Cai, Q.M. Lu, Parallel PIC code using Java on PC cluster, in: Proceedings of 4th International Conference/Exhibition on High Performance Computing in Asia-Pacific Region, Beijing, China, May 2000, pp. 495–500.
[24] T.R. Halfhill, Heating up Java, IBM Research Magazine 36 (4) (1998).