JPP: A PARTICLE TRACKING CODE WITH JAVA

Ryoichi Hajima*,
Dept. Quantum Eng. & Systems Sci., University of Tokyo,
Hongo 7-3-1 Bunkyo-ku, Tokyo 113-8656 Japan

Abstract

A computer language Java is a possible solution for the standardization of accelerator software to improve software productivity and realize common user interface. In this paper, JPP: a particle tracking code written in Java is presented as an example of Java application for accelerators. We discuss the advantage and disadvantage of developing scientific software with Java and investigate the performance of Java code in comparison with C code.

1 INTRODUCTION

In this section, we glance over the trend of software development and its application for accelerators and suggest that Java is a possible solution for the standardization of accelerator software, which improves software productivity and realizes common user interface.

1.1 Software for Accelerators : Development and Utilization

Computer software is widely used in accelerator field for modeling, designing, controlling, data acquisition and so on. A lot of programming languages and scripts have been suggested to construct the software. Computer hardware for these application is also variety from super computers to personal machines. Even a palm-top PC will be used as a personal terminal for scientific application in near future.

This variety of software and hardware confuses both programmers and users. Programmers often spend much time to translate their software from one platform to another platform, and users must learn minor difference existing in software translated from another platform. This is the reason why we desire the standardization of computer software for accelerators.

A programming language Java is a possible solution for the standardization and has already been utilized widely in industrial applications, because Java has a lot of fascinating features, which are architecture-neutral, tightly specified, robust and secure, and so on [1]. In scientific scene, however, it has been considered that a computer language running on an interpreter such as Java is not acceptable due to its poor performance.

In the present study, we describe a charged particle tracking code written in Java to discuss the advantage and disadvantage of Java in scientific applications, and investigate the validity of existing technologies to speed up Java.

* e-mail: hajima@q.t.u-tokyo.ac.jp

2 JAVA FOR NUMERICAL SIMULATIONS

In this section, we discuss the advantage and disadvantage of Java in scientific applications.

2.1 Advantage

2.1.1 Portable

Portability is the first distinguished feature of Java. Once we compile Java source code into byte code, the byte code is architecture-neutral and executable on any machine and any operating system as far as Java virtual machine (Java-VM) is available.

The portability is also a matter of concern for a program developer, because it promises freedom in choice of developer's environment. He can write and debug a code efficiently with his favorite editor on his familiar computer environment.

2.1.2 Object-Oriented

Java is object-oriented programming language and can be used to make class libraries for accelerator software, which have been constructed with C++.

2.1.3 Robust and Secure

Java provides a robust and secure methods to compose a program. For example of the robustness is that Java has no pointer data types defined in C and C++. The pointer is sometimes useful and permits us to make a tricky routine, but it has potential risk to cause unexpected memory destruction. The simple memory management model in Java releases us from run-time errors caused by abusing of memory management.

2.1.4 User Interface

Providing a common user interface is helpful for application users to learn the usage of new application without confusion. Java has its own graphical package AWT (Abstract Window Toolkit) to realize common user interface on different platforms.

2.1.5 Multithreading

We can write multithreaded applications with Java. Multithreading provides better interactive responsiveness and real-time behavior in interactive applications. Writing a multithreaded program is also enable us to scale up the performance of numerical applications, if the program is executed on a PC of multi-CPU.

2.2 Disadvantage

2.2.1 Performance

Performance of a program, computing speed, is matter of utmost concern to discuss the superiority of computer language for numerical simulations. It has been considered that Java has great disadvantage on this point, and is not acceptable for a numerical simulation. When we execute a Java program, Java VM interprets Java byte code and passes it to the CPU, in short words, Java is running on an interpretor. Interpreting byte code is generally much slower than executing native code directly. This is the reason why Java has only poor performance in numerical simulations.

People in Java, however, have made successive effort to improve the performance of Java and several solutions are becoming available, which are JIT, native compiler, native method, HotSpot VM.

3 SPEED-UP OF JAVA

As seen in the last section, improvement of performance, that is speed-up of calculation, is a key issue to make a numerical simulation with Java. In this section, we discuss some technologies improving the performance of Java.

3.1 JIT

The most popular way to speed up Java nowadays is JIT (Just-In-Time compiler). It is one of standard features supported by popular Java virtual machines such as JDK, Netscape Communicator, Internet Explorer. Just-in-time compiler translates whole of Java byte code into optimized native code on-the-fly and executes this native code, then the performance can be improved much better than a classical Java-VM, an interpreter. Since a JIT-VM reads a Java byte code and executes it as a classical-VM does, the portability of byte code is completely satisfied. The start up time overhead in the translation from byte code into native code is not a matter in numerical simulations, because calculation time is generally much longer than the overhead.

3.2 Native Compiler

Another straight way to speed up Java is using a native compiler, which translates byte code into native code and saves it as an executable file. One can directly run the executable file without Java VM. The code optimization in native compiler is superior to JIT-VM, because native compiler can spend more time for the optimization. The executable file is not architecture-neutral and the portability of Java is lost.

3.3 Native Method

If a specific routine in a Java program spends a large part of calculation time, we can save calculation time by rewriting this time consuming routine in C or Fortran and linking it to Java byte code. This is called as native method. In JDK (Java Development Kit), native method is implemented as JNI (Java Native Interface). The portability is lost as well as native compiler.

3.4 HotSpot VM

A new Java virtual machine named HotSpot is under development and released soon by Sun Microsystems [2]. The HotSpot VM includes several advanced technologies: adaptive optimization, generational garbage collection, hot spot detection and so on. It has been reported that the performance of HotSpot VM is three times as fast as the current JIT VM in JDK for typical benchmark problems [3].

4 DESCRIPTION OF JPP

A charged particle tracking code written in Java has been developed as an example of Java application for accelerators and the efficiency of existing technologies to speed up Java has been investigated. In this section, we describe the outline of the particle tracking code: JPP.

4.1 Class Hierarchy

The class hierarchy in JPP is designed as shown in figure 1. The root class is AccSystem.class which stores global parameters for calculations such as fundamental RF frequency, the number of active particles, time step for tracking. These parameters are kept as private variables to prevent unexpected overwriting of values in the simulation, but they can be referred from all the subclasses through public methods which return the value of private variables.

We prepare AccElement.class as a prototype of accelerator element and a class for each element such as drift and bending magnet is defined as a subclass of AccElement.class. It is easy to define a new subclass for a new accelerator element.

A routine for space charge calculation is defined in SpaceCharge.class which has several subclasses for various space charge models.

In particle tracking, each particle is treated as an instance of Particle.class, where variables of motion are stored as instance variables.

We also define ParticleSorce.class for a particle generator which initializes particle coordinates in six-dimensional phase space of motion according to input command. Another class, Jpp.class is prepared for the main routine of particle tracking.

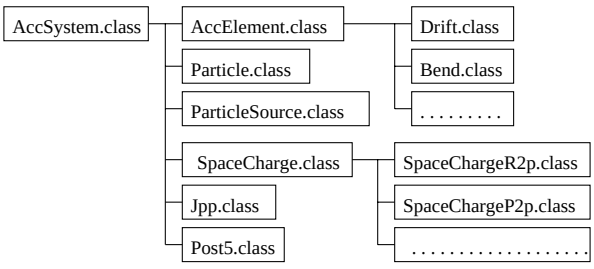

Figure 1: Class hierarchy in JPP.

4.2 Instance Methods of Accelerator Elements

All the accelerator elements defined in JPP have three basic instance methods to describe particle motion in the element, which are *inject()*, *eject()* and *advanceParticle()*. These methods are defined in AccElement.class and overridden in subclasses at need.

4.2.1 inject()

This method is called when a particle injects into the element and performs necessary procedures at the entrance of element. It checks if an injected particle exists inside the aperture. When the particle is out of the aperture, it is rejected from the active particle list. Some elements requires another procedure, for example, transverse momentum of the particle is modified at the entrance of a bending magnet with the fringe field and the edge focus.

4.2.2 eject()

This method is called when a particle ejects from the element. The particle coordinates in six-dimensional phase space at the exit of element is stored in output buffer of the element. The particle is, then, passed to the next element. In a bending magnet, the particle momentum is changed before the coordinates are recorded.

4.2.3 advanceParticle()

This method describes how a particle moves in the element. Every subclass of accelerator element has its own procedure for this method, because it is an intrinsic method characterizing the element.

4.3 Space Charge Calculation

Several subclasses are prepared for space charge calculation. Three different space charge models are implemented in JPP, which are ring-charge model known as SCHEFF in PARMELA [5], point-charge model [6] and line-charge model [7]. All the models are also available as native method with JNI (Java Native Interface), where time-consuming part of space charge routine is written in C and compiled into native code.

4.4 Input and Output File

JPP reads an input file in PARMELA-like format, where each line consists of one command keyword and several parameters following the command. Each command represents an accelerator element or simulation condition such as space charge model.

All the calculation results of JPP are recorded in a single output file, which contains the simulation parameters and particles coordinates at the exit of each accelerator element in binary format.

4.5 GUI/CUI Post Processor

A post processor POST5 is provided as a part of JPP to visualize simulation results and extract characteristic quantity such as beam emittance and envelope from an output file. The post processor POST5 is written in Java and all the GUI components are realized by java.awt (Abstract Window Toolkit), which is the standard API to provide graphical user interfaces in Java programs. Figure 2 shows a screen shot of POST5.

Character based interface is also available in POST5 with command line options, which is useful for quick checking of a specific quantity or running successive calculations in a batch job with logging a brief report of simulation results.

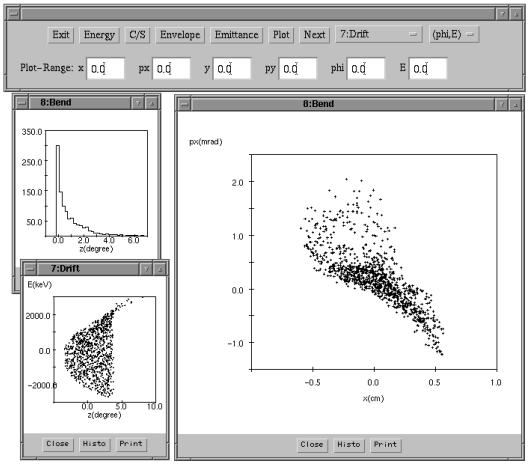

Figure 2: A screen shot of POST5 post processor.

5 EXAMPLES AND BENCHMARKS

Two examples of particle tracking have been made as benchmarks to check the validity of the simulation code and to evaluate the performance of Java code in comparison with C code.

5.1 Linear Emittance Growth in Simple Drift

Space charge calculation is one of essential routines in a charged particle tracking code. Calculation of linear emittance growth is a good example to check the accuracy of a space charge routine. When an electron bunch travels along simple drift space without external field, growth of transverse emittance occurs due to the self field of space charge. This emittance growth exists even if the bunch has uniform radial distribution, that is a hard edge cylinder, because the bunch has longitudinal discontinuity which causes nonuniform radial field depending on the longitudinal position in the bunch. In case that the electron bunch does not change its transverse size and the emittance growth mainly arises from transverse momentum fluctuation due to self field, the emittance grows linearly with drift length and can be calculated analytically. The linear growth of normalized trans-

verse emittance for an electron bunch having uniform transverse distribution is given by [4]

$$\Delta\varepsilon_n = \frac{I_p S}{4 I_a \gamma^2 \beta^2} G(L/a) \quad , \tag{1}$$

where I_p is peak current, $I_a = 17kA$ is Alfvén current, S is drift length and $G(L/a)$ is geometric function and given by $G(L/a) = 0.556$ for long Gaussian beam.

The linear emittance growth has been calculated by JPP with ring-charge model and the following parameters: $E = 5.11MeV$, $I_p = 220A$, $\sigma_z = 7.1cm$ and $r = 1cm$. The obtained emittance growth shows good agreement with analytical solution (fig. 3). Slight deviation of the numerical solutions from the analytical curve for long drift is considered as the result of radial expansion which introduces nonlinear effect in emittance growth. The expansion of beam radius is about 2% after drift of 60cm in this calculation.

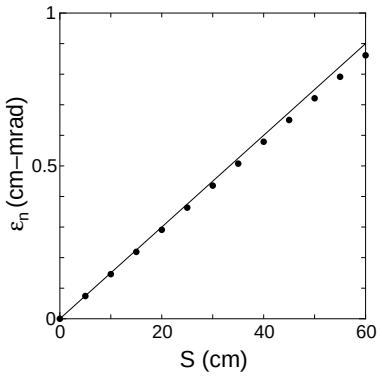

Figure 3: Calculation of linear emittance growth: analytical solution (solid line) and numerical results (dots).

5.2 Bunch Acceleration

Another example is bunch acceleration in RF cells. JPP reads RF field data prepared by SUPERFISH and calculates time varying electromagnetic filed at arbitrary position during the cells by interpolating field value at 2-D cylindrical grid. In this example, short electron bunch, $\sigma_z = 3.5mm$ and $r = 2mm$ is accelerated by 1300MHz 9-cell cavity, where accelerating RF phase is chosen so that maximum energy gain is obtained. Figure 4 shows obtained transverse phase plot and energy spectrum after the cells.

5.3 Performance Benchmark

The performance of JPP is compared with another particle tracking code which has the same numerical procedures as JPP but written in C. We also investigate how much the performance of Java program can be improved with JIT and native method.

In the benchmark, we use a personal computer: Pentium-Pro 200MHz, 256kByte L2 cache, 128MByte memory, working on Solaris 2.5.1. Java code is developed and executed on JDK–1.1.6 and C code is compiled by "ProCompiler C 3.0.1" with full optimization.

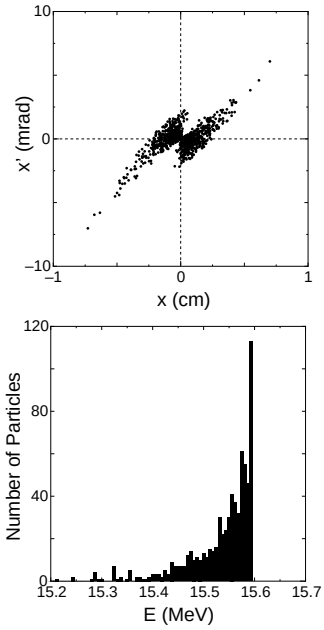

Figure 4: Results of bunch acceleration.

The performance of particle tracking code is measured by whole elapsed time for each run and three different input data are tested:

(1) bunch acceleration with ring-charge model
This example contains two heavy numerical routines: the calculation of space charge and RF field. The parameters are chosen as : the number of particles $N_p = 1000$, the number of time steps $N_s = 1754$, the number of 2-D cylindrical gird mesh for the space charge routine $(r, z) = (10, 30)$.

(2) simple drift with ring-charge model
In this case almost calculation time is spent by space charge routine. The parameters are $N_p = 500$, $N_s = 3000$, $(r, z) = (10, 30)$.

(3) simple drift with point-charge model
This is similar to the second example, but space charge effect is calculated from repulsion force between all the pair of two particles. This point charge model is quite heavy calculation. The parameters are $N_p = 500$, $N_s = 400$.

Table 1 shows the result of performance measurement, where T_a is whole elapsed time in second, T_s is time for space charge calculation, T_c is time for copying object between Java and C in native method and R is relative elapsed time in comparison with C.

The performance of Java with classical interpretor is really bad and the elapsed time is more than 16 times of C code in the worst case. It is also confirmed that JIT improves the performance of Java code greatly and the elapsed time is reduced to be two or three times of C code.

The performance of Java can be still improved, if time consuming routine is replaced by native compiled code. In our benchmark, space charge routine has been replaced by native method written in C and the performance has been surely improved as seen in table 1. Native method in JPP

Table 1: The result of performance measurement.

Bunch Acceleration (ring-charge)

	T_a	T_s	T_c	R
Java (interpretor)	959	806	–	11.7
Java (JIT)	203	156	–	2.5
Java (JIT + native)	166	119	80	2.0
C	82	50	–	1.0

Simple Drift (ring-charge)

	T_a	T_s	T_c	R
Java (interpretor)	1238	1204	–	16.7
Java (JIT)	217	205	–	2.9
Java (JIT + native)	98	86	42	1.3
C	74	70	–	1.0

Simple Drift (point-charge)

	T_a	T_s	T_c	R
Java (interpretor)	601	594	–	8.7
Java (JIT)	157	153	–	2.3
Java (JIT + native)	61	58	6	0.88
C	69	68	–	1.0

is implemented by JNI (Java Native Interface) which Sun Microsystems provides as the standard implementation of native method.

Although we expected that space charge calculation in Java with native method is as fast as C code, Java does not catch up with C in some cases. This is because that native method introduces another overhead to copy object array between Java and C. In our calculation with native method, the array of particle object stored in Java working memory is transfered into memory for native method one by one before space charge routine starts and vice versa after space charge calculation is completed. This procedure may become a severe overhead, if we use large number of particles beyond the size of cache. Instead of copying object array one by one, native method could be implemented in other ways in which one can refer and rewrite Java working memory from native method or copy object array at once. However, JNI prohibits these operation and only permits to copy object array one by one, because it is the most secure method which never makes unexpected memory destruction. Performance of native method is, therefore, determined by overhead of copying memory and speed-up due to native code.

Native method contributes to the improvement of performance particularly in the third example of our benchmark, where space charge routine is quite heavy and spends much more time than the memory copy. It has also been confirmed in another study that the more complicated space charge calculation, noninertial space charge force in circular motion of electron bunch, can be available with JPP and native method [8]. In the case of bunch acceleration, on the contrary, native method saves calculation time little, because the number of particle is relatively large and RF field calculation is still conducted by Java byte code.

In conclusion of the benchmark, it is found that JIT significantly improves the performance of the particle tracking code written in Java without loosing the portability, the Java code, however, still requires calculation time two or three times as great as optimized C code in typical problems.

6 CONCLUSIONS

A charged particle tracking code has been developed with Java to investigate possible standardization of accelerator softwares with Java. It has been confirmed that the particle tracking code written in Java has several outstanding features: it is executable on a lot of different platforms without recompiling, providing unified graphical user interface, easy to maintain or extend due to sophisticated class hierarchy and robustness. The performance of the particle tracking code is inferior to optimized C code and it requires calculation time two or three times as large as C code, even if we use Java-VM with JIT (Just-In-Time compiler). The performance, however, will catch up with C code by upcoming HotSpot VM.

7 REFERENCES

[1] J. Gosling and H. McGilton, "The Java Language Environment A White Paper", <http://java.sun.com/docs/white/langenv/>(1996).

[2] D. Griswold, "A White Paper About Sun's Second Generation Java Virtual Machine", <http://java.sun.com/products/hotspot/index.html>(1998).

[3] D. Griswold, "JDK Software: Performance Technologies", JavaOne '98 (1998).

[4] M.E. Jones and B.E. Carlsten, in Proc. 1987 Particle Accelerator Conf., pp.1319-1321.

[5] P. Lapostolle et al., Nucl. Inst. Meth. **A379**, pp.21–40 (1996).

[6] K.T. McDonald, IEEE Trans. Electron Devices, **35**, pp.2052–2059 (1988).

[7] B.E. Carlsten et al., Nucl. Instr. and Meth. **A304**, pp.587–592 (1991).

[8] R. Hajima and E. Ikeno, "Numerical analysis of shielded coherent radiation and noninertial space-charge force with 3-D particle tracking", to be published in Nucl. Instr. and Meth. A.