PERFORMANCE EVALUATION OF PARTICLE TRACKING SIMULATION WITH JAVA

Ryoichi Hajima, Univ. Tokyo, Hongo 7-3-1, Bunkyo-ku, Tokyo, 113-8656 Japan

Abstract

Programming language Java promises a possible solution for standardization of accelerator software: accelerator modeling, designing, controlling, operating, logging and so on. Performance of Java is a key issue in these applications, if we replace existing simulation codes and class libraries into Java. In the present study, performance of a charged particle tracking code with Java which includes space charge calculation is evaluated in comparison with C code. It has been found that the performance of Java is significantly improved by JIT compiler and comparable to C code.

1 INTRODUCTION

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 proposed to develop the software. Computer hardware for these application is also variety from super computers to personal machines. This diversity of computer software and hardware often confuses both programmers and users. Programmers spend much time to translate their software from one platform to another platform, and users must learn different user-interface on various platforms. This is the reason why we desire the standardization of computer software for accelerators.

A programming language Java provides a possible solution for the standardization, because Java is completely platform independent [1] [2]. 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 evaluate the performance of a particle tracking code written in Java for the further discussion of standardization of accelerator software.

Although there is another problem to be discussed for numerical simulation with Java, which is related with inadequate specification of floating-point arithmetic in Java [3], we do not treat such a problem and focus on the performance issue in the present study.

2 DESCRIPTION OF THE PARTICLE TRACKING CODE

In this section, we describe the outline of a Java particle tracking code: JPP, which has been developed as an example of Java application for accelerators [4].

2.1 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 during 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. Once we construct this class hierarchy, it is easy to define a new subclass for a new accelerator element with full use of inheritance from superclass.

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.

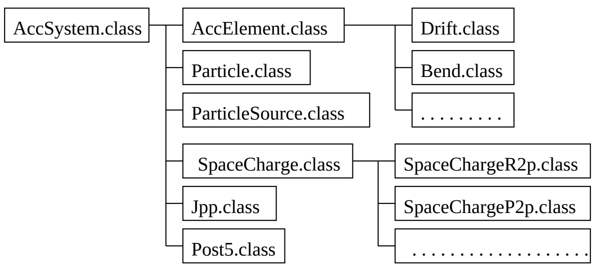

Figure 1: Class hierarchy in JPP.

2.2 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.

2.3 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 interface 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.

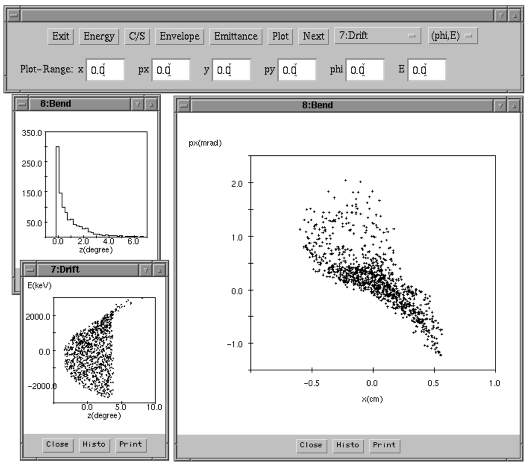

Figure 2: A screen shot of POST5 post processor.

3 SPEED-UP TECHNIQUE FOR JAVA

Improvement of performance, speed-up of calculation, is a critical issue to make a numerical simulation with Java. In this section, we discuss technique 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, the portability of byte code is completely guaranteed.

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 generally superior to JIT-VM, because native compiler can spend longer time for the optimization. However, the executable file is not architecture-neutral and the portability of Java is lost.

3.3 Native Method (JNI)

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. In our particle tracking, native method is applied to space charge routines, which consume more than 90% of calculation time.

4 BENCHMARK TESTING

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 can be improved with JIT and JNI.

In the benchmark, we use two platforms:

(a) Pentium MMX 266MHz, 256kByte L2 cache, 64MByte memory, working on Windows-98. Java code is developed and executed on JDK-1.2-V with option “-O”, and C code is compiled by “egcs-2.9.1.57” with option “-m486 -O4”.

(b) Pentium-Pro 200MHz, 256kByte L2 cache, 128MByte memory, working on Solaris 2.5.1. Java code is developed and executed on JDK-1.2-01-dev05-fcsK with option “-O” and C code is compiled by “ProCompiler C 3.0.1” with option “-fast -xO4 -xpentium”.

The performance of particle tracking code is measured by whole elapsed time for each run and three different input data, which are the same as the previous study [4], 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 most of 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 $N_p = 500$, $N_s = 400$.

Tables 1 and 2 show the result of performance measurement for two platforms, where T is whole elapsed time in second and R is relative elapsed time in comparison with C.

Table 1: Performance measurement on Windows 98.

Bunch Acceleration (ring-charge)

	$T(sec)$	R
Java (interpretor)	491.03	6.55
Java (JIT)	60.15	0.803
C	74.92	1.0

Simple Drift (ring-charge)

	$T(sec)$	R
Java (interpretor)	322.85	6.36
Java (JIT)	43.88	0.865
C	50.75	1.0

Simple Drift (point-charge)

	$T(sec)$	R
Java (interpretor)	611.49	7.70
Java (JIT)	56.13	0.707
C	79.37	1.0

Table 2: Performance measurement on Solaris.

Bunch Acceleration (ring-charge)

	$T(sec)$	R
Java (JIT)	160.63	2.28
Java (JIT+JNI)	71.69	1.02
C	70.58	1.0

Simple Drift (ring-charge)

	$T(sec)$	R
Java (JIT)	82.71	2.42
Java (JIT+JNI)	36.00	1.05
C	34.25	1.0

Simple Drift (point-charge)

	$T(sec)$	R
Java (JIT)	109.00	2.63
Java (JIT+JNI)	40.42	0.976
C	41.42	1.0

On Windows-98 platform, it is confirmed that JIT improves the performance of Java greatly and JIT is faster than C code in all the examples. This is not a surprising result, but reasonable to understand as follows. The performance of Java in general applications may degraded by several factors, which are thread synchronization, garbage collection and overhead for object creation. In our particle tracking these are none of issues, because all the objects, particles and accelerator components, are generated at once in the beginning of simulation. Another overhead due to the translation from byte code into native code at the start-up of JIT is also negligible in numerical simulations, because calculation time is generally much longer than the overhead. Performance of particle tracking in Java and C, therefore, does not depends on language specification, but determined by efficiency of compiler optimization.

On Solaris platform, Java with JIT shows worse performance than C code and we need JNI to catch up with C. Detail analysis with a profiling tool shows that the difference of performance between JIT and C comes from a single "for-loop" nested in quadruplicate. This may be because that Java VM we used on Solaris is still beta version.

When native method with JNI is applied to particle tracking, overhead caused by operation of object array becomes a matter of concern. Since Java has only the array of object reference, copy of large object array between Java and C in JNI must be made one by one and results in degradation of the performance [4]. This overhead, however, has been almost cleared in JDK-1.2 and JNI shows fine performance comparable to C.

5 CONCLUSIONS

Performance evaluation of a charged particle tracking code written in Java has been made. It has been found that inevitable performance degradation due to Java language specification, which includes garbage collection, object creation and operation of object arrays, is not a critical issue for particle tracking, and Java with JIT shows excellent performance comparable to C on Windows 98 platform.

6 ACKNOWLEDGMENT

The author would like to thank H.Nishimura for helpful discussion.

7 REFERENCES

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

[2] H.Nishimura, "Can Java Replace C++ on Windows for Accelerator Control?", in this proceeding.

[3] W.Kahan and J.D.Darcy, "How Java's Floating-Point Hurts Everyone Everywhere", ACM 1998 Workshop on Java for High-Performance Network Computing (1998). <http://www.cs.berkeley.edu/ wkahan/JAVAhurt.pdf>

[4] R. Hajima, "JPP: A Particle Tracking Code With Java", to be published in Proc. International Computational Accelerator Physics Conference (1998).

[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).