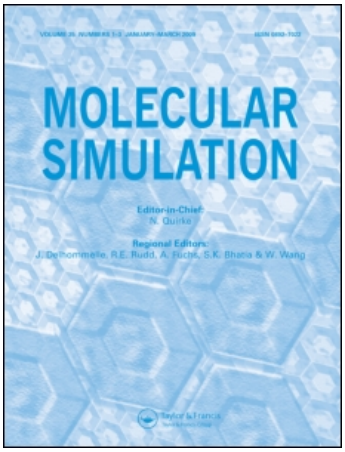

Molecular Simulation
Publication details, including instructions for authors and subscription information:
http://www.informaworld.com/smpp/title~content=t713644482

A Molecular Dynamics Computer Simulation Performance Comparison of Java Versus C

M. J. D. Mallett[a]
[a] The School of Physical Sciences, University of Kent, Canterbury, Kent, UK

To cite this Article Mallett, M. J. D.(2001) 'A Molecular Dynamics Computer Simulation Performance Comparison of Java Versus C', Molecular Simulation, 26: 6, 417 — 422
To link to this Article: DOI: 10.1080/08927020108024514
URL: http://dx.doi.org/10.1080/08927020108024514

Molecular Simulation, 2001, Vol. 26, pp. 417–422
Reprints available directly from the publisher
Photocopying permitted by license only

A MOLECULAR DYNAMICS COMPUTER SIMULATION PERFORMANCE COMPARISON OF JAVA *VERSUS* C

M. J. D. MALLETT

The School of Physical Sciences, University of Kent, Canterbury, Kent, UK

(Received July 2000; accepted September 2000)

The relative performance of molecular dynamics (MD) computer simulations of fluids written in ANSI C is compared to that achieved by a comparable program written in Java. The performance of the Java program is shown to be dependent upon its runtime environment. The Java Runtime Environment (JRE) from the Java Development Kit (JDK) 1.2 provides a Just-In-Time (JIT) compiler option on Solaris and Windows 95 platforms which decreases the execution time by approximately 4–10x compared to the standard Java interpreter. The compiled Java implementation of the MD computer simulation runs between 30–100% slower, depending on the platform, compared to the equivalent C implementation. The stability of the two simulations, as measured by conservation of energy is shown to be identical to within $\sim 1\%$ over 10^5 time steps.

Keywords: Java; Molecular dynamics; Computer simulation

INTRODUCTION

The digital computer provides the capability to undertake numerically intensive calculations in a relatively short space of time. This has allowed the development of computer simulations based on theoretical models of real systems. Calculating the interactions between the component parts of any representative model is computationally expensive. It is only with the advent of modern, high speed computers that models which can be said to be representative of the real system have been developed. In the field of molecular dynamics (MD) [1, 2] the requirement is to follow the motion of particles or molecules in a computer simulation as their configuration evolves through time. The validity of the simulation is dependent upon how

well the interactions are modelled and how many particles are incorporated into the model. When a large number of particles is used (> 1000) the computational requirement to follow the behaviour of the system is very large. Consequently it becomes advantageous to use a fast computer and to ensure that the computer simulation operates efficiently.

There is considerable debate as to the "best" computer language to use when writing MD code. Traditionally most researchers use FORTRAN, both from a historical viewpoint but also because FORTRAN compilers have been well developed over the years to provide fast efficient programs. An alternative language used by a significant minority of researchers is C/C++. The efficiency and execution speed of programs compiled from either FORTRAN or C/C++ is comparable. The most notable newcomer in the world of computer languages is Java. The intention of this communication is to determine the feasibility of using Java as a programming language for large-scale MD simulations.

The Java programming language has been developed by Sun Microsystems Inc. [3]. It was originally designed to be implemented in embedded systems, but with the evolution of the Internet and in particular the World Wide Web it is now predominantly used to enhance the appearance and functionality of hypertext web pages. However the Java language is a fully functional programming language in its own right with essentially the same functionality as the mainstream computer languages (*e.g.*, FORTRAN, C/C++, Pascal, *etc.*). It also has the distinct advantage of being designed from the outset to operate efficiently over a distributed network of computers using a Uniform Resource Locator (URL) as the means of accessing objects on another computer. This offers the possibility of performing distributed computations over a heterogeneous network of computers.

The two major requirements of a computer language when applied to MD are the speed of execution and the precision of calculation. A typical MD program is often required to simulate the motion of a large number of particles, typically 10^3 over a large number of time steps, typically $> 10^6$. The execution time of the program is frequently of the order of days or weeks. It then becomes a matter of necessity to ensure that the simulation is not speed restricted by a poor choice of programming language and compiler. It is also as important that the stability of the simulation, for instance in terms of the conservation of total energy, is not compromised by the data precision of the computer language. The ability of the Java programming language to address these two criteria needs to be ascertained to determine its suitability as a scientific programming language for MD simulations.

RESULTS

A simple MD program, written in C, for soft spheres in a unit cell with periodic boundary conditions was already available. The interaction between spheres was calculated using a shifted-force, Lennard-Jones 6–12 potential [4, 5]. The equations of motion were computed using a 5th order Gear predictor–corrector algorithm.

It was straightforward to write a directly analogous program in Java. In particular most of the syntax of C and Java is identical, which means that not only are C programs relatively easy to convert to Java, but also the program functionality should be almost exactly equivalent.

Two computing platforms were used to determine if there might be any architecture dependence between the execution of a C or Java program. The computers used were an Intel Pentium powered PC running either Windows 95 or Linux, and a Sun SparcStation 4 running Solaris 7. The C source code was compiled using a version of Gnu C Compiler (GCC) 2.7 on both platforms; on the Intel platform the C source code was compiled under the Linux operating system because of the easy availability of ports of GCC to it. The Java source code was compiled using the Java compiler supplied in the Java Development Kit (JDK) version 1.2. This particular version of the JDK supplies a Just-In-Time compiler (JIT) for both platforms. The JIT compiles the byte-code produced by the Java compiler into a native form specifically for that platform. Java programs which are recompiled using a JIT will execute significantly faster than those which use the standard Java interpreter. The Solaris version uses the "sunwjit" compiler, supplied by Sun, while the Windows 95 version uses the "symcjit" compiler supplied by Symantec [6]. It was decided to use the Windows 95 operating system for the comparative Java simulations because the most effort in developing an efficient JIT has been directed towards this operating system.

The simulations were run using model systems containing 108 particles and 256 particles. The particles had been previously melted from a face-centre cubic formation into a random configuration. The simulations were run for 10^5 time steps to provide a representative number of steps for the two models to be compared. The speed of execution and the conservation of energy were monitored.

The execution speed comparison of Table I shows the relative performance of the two compiled languages. On both platforms the C compiled code runs faster than both the Java interpreter and the Java compiled code. On the Sparc platform the Java interpreter was $> 800\%$ slower than the

TABLE I Comparison of execution speeds between GCC and Java in seconds; where Java[1] uses the native Java interpreter and Java[2] uses the Java JIT complier. The number in brackets indicates the uncertainty in the value

Operating system	*Number of particles* = 108			*Number of particles* = 256		
(*computer*)	*GCC*	*Java*[1]	*Java*[2]	*GCC*	*Java*[1]	*Java*[2]
Solaris 7 (Sparc)	13 (1)	112 (2)	26 (2)	70 (4)	579 (2)	136 (6)
Windows 95 (Intel)	–	234 (8)	21 (2)	–	1192 (8)	113 (5)
Linux 2.0 (Intel)	15 (1)	–	–	88 (5)	–	–

TABLE IIa Ratio of stability of energy conservation, number of particles = 108

Language	*No. of steps* = 10^3	*No. of steps* = 10^4	*No. of steps* = 10^5
GCC	4.5 e 10^{-7}	6.4 e 10^{-6}	4.8 e 10^{-5}
Java	4.5 e 10^{-7}	5.4 e 10^{-6}	4.8 e 10^{-5}

TABLE IIb Ratio of stability of energy conservation, number of particles = 256

Language	*No. of steps* = 10^3	*No. of steps* = 10^4	*No. of steps* = 10^5
GCC	3.7 e 10^{-6}	3.4 e 10^{-5}	3.4 e 10^{-4}
Java	3.7 e 10^{-6}	3.6 e 10^{-5}	3.4 e 10^{-4}

equivalent C code while the Java compiled code was only about 100% slower. The Intel platform showed an even greater difference. The interpreted Java code was > 1300% slower while the compiled Java code ran only approximately 30–40% slower. The Java interpreter was more efficient on the Sparc platform, whilst the Intel platform handled compiled Java code more efficiently. On the Sparc platform the compiled Java code ran ~ 4.3 times faster than the interpreted code and on the Intel platform the compiled code ran > 10 times faster than the interpreted code. This is more likely to indicate that the Symantec JIT compiler on the Intel machine produces more efficient native code than does the Sun JIT compiler on the Sparc platform. The development of Java JIT compilers on both platforms is still at an early stage whereas C compilers should now work optimally on almost any platform.

In Tables IIa and b the stability of energy conservation is compared for simulations running under the different compiled languages. No attempt was made to periodically rescale the total energy of the simulations. This is usually achieved using a suitable thermostat to ensure energy conservation. The lack of energy rescaling provides a measure of how quickly the simulation will diverge from its initial thermodynamic state as the simulation evolves in time. The tables show the fractional change in total energy as a function of the number of time steps. As the number of particles in the

simulation increases, the number of calculations increases and there will be greater accumulated round-off error, leading to a divergence of the total energy from its initial value. The results show that there is no significant difference between the accuracy of the simulation compiled in C to that compiled in Java. In both cases the floating point variables were of type double. The Java programming language uses the IEEE 754 standard [7] for defining the way floating point numbers are dealt with. In the case of variables of type double the variables are stored as 64 bit numbers, this is the standard Java implementation and is used on every platform that implements a Java environment. The results of a calculation in a Java program will therefore be the same on every platform on which it is executed. The definition of variables in C can be more flexible. In the original Kernighan and Ritchie [8] version of C there were no set definitions for the byte allocation of the data types. They were platform dependent, leading to the possibility of different machines producing different calculated results from a common source code. The floating point data types in GCC follow the IEEE 754 standard.

During the implementation of the MD simulation in Java it was noticed that an unexpected error was accumulating in the floating point counter which measured the elapsed time in reduced units of Verlet. The floating-point time counter was of type double, initially with the value 0.0, and was incremented by an amount delta ($=0.02$) at each time step. After a number of increments the counter diverged away from the expected value. This is a consequence of the finite representation of data on a digital computer. A 64 bit double precision number in the IEEE 754 standard has 52 bits allocated for the fractional part allowing ~ 16 significant figures to be represented. This means that any floating-point number which is not an exact power of 2 cannot be accurately represented. This is a well known aspect of floating point representations. However in most modern compilers provision has been made to allow floating-point numbers to be represented in a form suitable for the precision of the calculation required. The IEEE 754 standard defines four allowed rounding conditions that can be applied to floating point numbers, (i) round to nearest, (ii) round to $+$infinity, (iii) round to $-$infinity, (iv) round to zero. In the Java implementation of the IEEE 754 definition for floating point numbers the *round to nearest* technique is used [9]. This changes the way certain floating-point numbers are handled in Java compared to the behaviour of the reference language, GCC. It was necessary to modify the program to take account of this when using floating-point numbers for counters when the floating-point representation of an integer is expected after a certain number of operations. The round off behaviour of

Java appears to have no effect on the stability of the simulation as shown in Tables IIa and b.

DISCUSSION

The decision to use the Java programming language for MD computer simulations can be justified for a number of different reasons. The language has a more advanced capability for exception handling than most of its competitors, it produces code which can be run on any platform with an implementation of the Java Runtime Environment (JRE), but possibly more importantly it offers the capability of easily implemented distributed computing over a heterogeneous network of computers. There are, however, definite disadvantages to using Java compared to a more established computing language, for instance C/C++ or FORTRAN, particularly in relation to the speed of execution of comparable source code. The speed disadvantage should become less marked as the Java JIT compilers improve in performance; though it is unlikely that the speed of a Java compiled program will match that of a native compiler due to the requirements of portability of the Java bytecode over all platforms. There would appear to be no penalty in using Java in terms of the long-term stability of the simulation as measured by its ability to conserve energy.

References

[1] Allen, M. P. and Tildesley, D. J. (1989). *Computer simulation of liquids*, Oxford.
[2] Haile, J. M. (1992). *Molecular dynamics simulation*, Wiley.
[3] Sun Microsystems Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA.
[4] Nicolas, J. J., Gubbins, K. E., Streett, W. B. and Tildesley, D. J. (1979). *Molec. Phys.*, **37**, 1429–1454.
[5] Powles, J. G., Evans, W. A. B. and Quirke, N. (1982). *Molec. Phys.*, **46**, 1347–1370; erratum, *Molec. Phys.*, **51**, 1511 (1984).
[6] Symantec Corporation, 10201 Torre Avenue, Cupertino, CA 95014 USA.
[7] IEEE Standard 754–1985 for Binary Floating-Point Arithmetic.
[8] Kernighan, B. W. and Ritchie, D. M. (1978). The C programming language, Prentice-Hall.
[9] Kahan, W. and Darcy, J. D., "How Java's floating-point hurts everyone everywhere", [http://www.cs.berkeley.edu/~wkahan/JAVAhurt.Pdf] (current as of July, 2000).