

A Java-Based Direct Monte Carlo Simulation of a Nano-Scale Pulse Detonation Engine

Darryl J. Genovesi (genovesi@psu.edu) and Lyle N. Long (lnl@psu.edu)

*Department of Aerospace Engineering
The Pennsylvania State University
233 Hammond Building
University Park, PA 16802*

Abstract. Recently, the Aerospace community has focused much of its effort on the development of micro-air vehicle technology. The treatment of problems at the micro-scale and nano-scale levels has traditionally been difficult, and one of the most challenging aspects of the design of these aircraft is the development of compact propulsion systems for them. Here, the pulse detonation engine is proposed as a means of propulsion for micro-air vehicles and nano-air vehicles and its performance is simulated using a DSMC flow solver. The major problem arising when attempting to implement the pulse detonation engine at such small length scales is the dominance of the wall effects inside the detonation chamber. To alleviate the loss of thrust due to wall effects, an adiabatic wall boundary condition was developed, ultimately affecting a thermodynamic environment that is more favorable for the formation of a detonation wave, while maintaining real-wall viscous effects. The chemistry and physics of the pulse detonation processes are then simulated in order to determine the minimum allowable combustion chamber dimensions to facilitate ample thrust for the micro-air vehicle and nano-air vehicle problems.

The code discussed in this paper is written in Java 2 and compiled using both GNU G++ v.1.0 and Sun Java Development Kit 1.3.1. Traditionally, FORTRAN has been used for the solution of scientific problems, but more recently, C and C++ have replaced FORTRAN in some cases. With an increase in computer power, however comes the ability to develop larger software packages, thus spawning the need for a more robust computational tool. The object-oriented structure of Java lends itself very well to the Direct Simulation Monte Carlo flow solver and to particle based flow solvers in general. In particular, the relatively complex particle sorting routine used in G.A. Bird's original DSMC flow solver [1] can be replaced by a much simpler one in an object oriented program environment. Although a similar object oriented environment exists in C++, Java's intrinsic multi-threading capability, which C++ lacks, makes the problem quite amenable to solution on multiple processors in a shared memory environment without the added complexity of MPI or OpenMP. Furthermore, Java's graphical capabilities can be particularly useful in monitoring the solution as it is averaged over many ensembles. The trade-off, of course, is a mild performance degradation as compared to FORTRAN and C/C++ on some computer systems. However, the difference in performance is found to be surprisingly small as compared to FORTRAN and, in some cases, Java even outperformed the FORTRAN 90 solution to an identical problem.

INTRODUCTION

The simulation of detonation processes using DSMC was first explored in one dimension by Long and Anderson [2] based on the unimolecular chemical reaction model of Dunn and Anderson [3] and found to give solutions that agree very well with the Chapman-Jouguet solution. Here, the simulation is extended to two dimensions and applied to the specific problem of the pulse detonation engine. Pulse detonation technology has been studied extensively as a means of propulsion, its major advantages being its simplicity (no turbo machinery) and its efficiency due to the high speed at which the combustion occurs as compared to a flame. The rapid energy conversion results in a process that is thermodynamically much closer to a constant volume

cycle than a constant pressure one and is therefore much more efficient [4]. Current research concentrates on the use of pulse detonation technology for much larger propulsion systems and continuum-based numerical solutions have been developed [5,6]. If the Pulse Detonation Engine is simulated on nano-length scales, the continuum model breaks down and a particle-based method such as DSMC must be employed.

The critical ignition temperature and cross-sectional area for formation and self sustained propagation of a detonation wave have been explored for several different fuel-oxidizer mixtures. With the exception of acetylene, a practical means of initiating detonations for traditional hydrocarbon-air mixtures, even at macroscopic length scales, has not yet been found [7]. The amount of energy required, and the rate at which this energy must be deposited into the system, exceeds the limitations of current technology. It has been shown [8] that the size of the detonation cells, the multidimensional packets of energy that form in and around a detonation wave during initiation, is directly related to the critical energy and the minimum chamber dimension. Typical cells sizes have been found to be around 1 cm and 30 cm for Acetylene and Methane-fueled detonations, respectively, suggesting that fuels such as methane will require a very large amount of energy input to the system in order to initiate a detonation [9]. In short, the practicality of actually implementing the PDE at nano-length scales depends on the ability to develop new fuels with much smaller detonation cell sizes. Although the physical restrictions involving these length scales as well as the minimum achievable time to initiation of a detonation wave will, in all likelihood, curtail the implementation of a PDE system at the length scales discussed in this report, the possibility exists for a slightly larger system, one that may be used to propel a very small aircraft but large enough that detonation initiation is possible. For such a device, the performance will obviously be highly dependent on wall effects. Therefore, multi-dimensional simulations are required and the extension to two dimensions follows, as described in full by [10].

JAVA AND DSMC

The object oriented structure of the Java programming language allows certain enhancements to be made to the traditional DSMC method, most importantly the use of dynamically allocatable particle arrays and Java multithreading. An attempt will be made to document the code development process in as much detail as possible so that the reader can make an accurate assessment of the importance of these features.

Dynamically Allocatable Particle Arrays

Intrinsic to Java is the Vector class, a method for creating one-dimensional, dynamically allocatable arrays. Vectors are initialized to a user-defined length upon instantiation. As the program runs, the length of the Vector can be increased or decreased in order to accommodate all the objects that must be held within it while minimizing the required amount of memory. Elements can be added and removed at any position in the Vector and these operations are performed seamlessly with very little programming required. A similar dynamically allocatable array subsystem could also be implemented easily in C++ using pointers.

The computational overhead associated with the implementation of the Vector class remains to be exactly determined, but preliminary indications are that the computational time required to perform Vector operations is minimal for computationally intensive applications such as DSMC. Although it might not be obvious, the use of the Vector class can greatly simplify the DSMC method by allowing the replacement of the traditional DSMC *sort* routine with a much simpler one. In order to motivate the discussion, note that in an object oriented approach to DSMC calculations, it would seem sensible for the domain object to have ownership of all of the cell objects, which in turn have ownership of a number of particle objects.

Since particles are constantly moving from one cell to another, a constant-length array of particles in each cell is not a possibility. The traditional means of dealing with this problem is to allow the domain object to have ownership of all cell and particle objects and the DSMC methodology is applied as it would be in a procedural programming approach. However, in order to take full advantage of the object oriented structure, it is proposed that the Vector class be used to store the particle objects inside of the cell objects, essentially moving the responsibility of carrying out the computations for the *move* and *collide* routines from the domain object to the cell objects. After the *move* routine is finished but before the *collide* routine is invoked, each particle is sorted by the domain object, and then redistributed to its respective cell object.

The new *sort* routine works as follows. First, the domain object loops over all of the cell objects. Each cell is asked to determine which of its particles moved to a new cell in the *move* routine. Those particles are returned to the domain object. The particle objects that were returned are then asked to provide their x, y and z coordinates, which are directly translated into a cell location. The particle is then added to the end of the Vector of particle objects in the cell in which the particle is now located and the Vector is resized if necessary. In turn, the particle is removed from the particle object Vector in the cell in which it previously resided.

Although there is no immediate performance advantage to using the new scheme, it is much more elegant than the traditional method. Significant gains are made insofar as code readability and maintainability. The original DSMC *sort* routine used a long series of complicated integer array computations, which were difficult for the average programmer to understand. The new method is quite straightforward and should be much easier to understand for someone who is new to DSMC methodology.

Parallelization Using Java Multithreading

DSMC solutions for large, complicated geometries may, in many cases, have impractically large memory and CPU time requirements. In this case, a domain decomposition strategy must be employed using MPI or some other parallelization language on large distributed memory systems. However, for the simple geometries discussed here, where the memory requirements are fairly low, a domain decomposition strategy is not required and a solution may be obtained equally as quickly using a much simpler parallelization strategy. The ensemble-averaging technique used to obtain an adequate number of samples motivates the development of a code that is nearly parallel already. That is, the problem is already broken up into many sub-problems at the highest level. All that remains to be done is to feed each of these sub-problems to a different processor and to get the main program to average all of the solutions together at the end of the computation. This is a perfect example of the parallelization of multiple serial jobs discussed in [11].

Java multithreading is a very convenient method for parallelization in shared memory environments. All of the multithreading capabilities are built into the language, thus eliminating the added complexity brought about by the use of MPI or OpenMP. The multithreading capabilities can be extended to work in distributed memory environments as well with Java Remote Method Invocation (RMI). Java multithreading and RMI work together quite well for single-client, single-server applications, but running parallel master-slave jobs is difficult at best. In order to run parallel java jobs on distributed memory machines, other means should be sought including a new message passing standard for Java. For the present time, however, the technology simply is not sufficiently developed to use for this DSMC application. Therefore, all parallel jobs were simply run on shared memory machines.

COMPUTATIONAL RESULTS

The DSMC code was set up to run a series of 2-D Pulse Detonation Engine cycles at an initial temperature of 300 K, using monatomic particles of mass 40 grams/mole for both the reactants and the products. A one-step chemical reaction model was used [3,2] with activation energy and heat release specified as 8.0 and 20.0 kcal/mole, respectively. All simulations were run using uniform cells of edge lengths equal to one half the initial mean-free path and initially containing 25 particles each over 40 ensembles for a total of 1000 samples per cell per time-step, on average.

The case of a detonation in a tube was first simulated using the specular (inviscid) wall boundary condition. Although it is not a complete representation of reality, a great deal of the physics of a detonation problem appears in this simple one-dimensional solution. Figure 1a. shows the species concentration of chemical reactants at a given time in the detonation simulation. A well-defined detonation wave is observed at approximately $2/3$ the length of the tube and is moving from left to right. The solid region to the right of the wave contains only reactants, while that on the left of the wave contains only reaction products. The wave is self-sustaining and moves from left to right through the domain at velocities in agreement with the Chapman-Jouguet theory and the computational results presented in [2].

The implementation of a real, viscous wall will now be considered. Since the current problem is characterized by very small length scales, the wall effects become the most important consideration in its treatment. Specifically, the viscous losses at the wall and the heat-transfer to the wall both serve to remove energy from the flow and could quench the detonation wave. In order to simulate an isothermal, viscous wall, the diffuse wall subroutine is implemented. First, the case is considered where the wall temperature is equal to the initial temperature of the reactants. The results of this simulation are given in Figure 1b., at the same point in time as for the specular wall case above. It observed that a series of chemical reactions takes place inside the ignition region, but the energy that is transferred to the cold wall prevents a detonation wave from forming. Without a detonation wave, there can be no thrust and the system fails. The detonation problem was run again, using the same viscous wall subroutine, but with a wall temperature ten times greater than the initial temperature of the reactants in attempt to reduce the energy loss and facilitate the initiation of a detonation wave. As seen in Figure 1c., at the same point in time, the detonation chamber is entirely filled with chemical reaction products. This scenario did not result from a wave propagating through the domain as desired, but rather from a nearly spontaneous combustion process, whereby transverse waves starting at the upper and lower walls met along the centerline of the duct, without inducing a bulk velocity in the axial direction. Again, no thrust is produced.

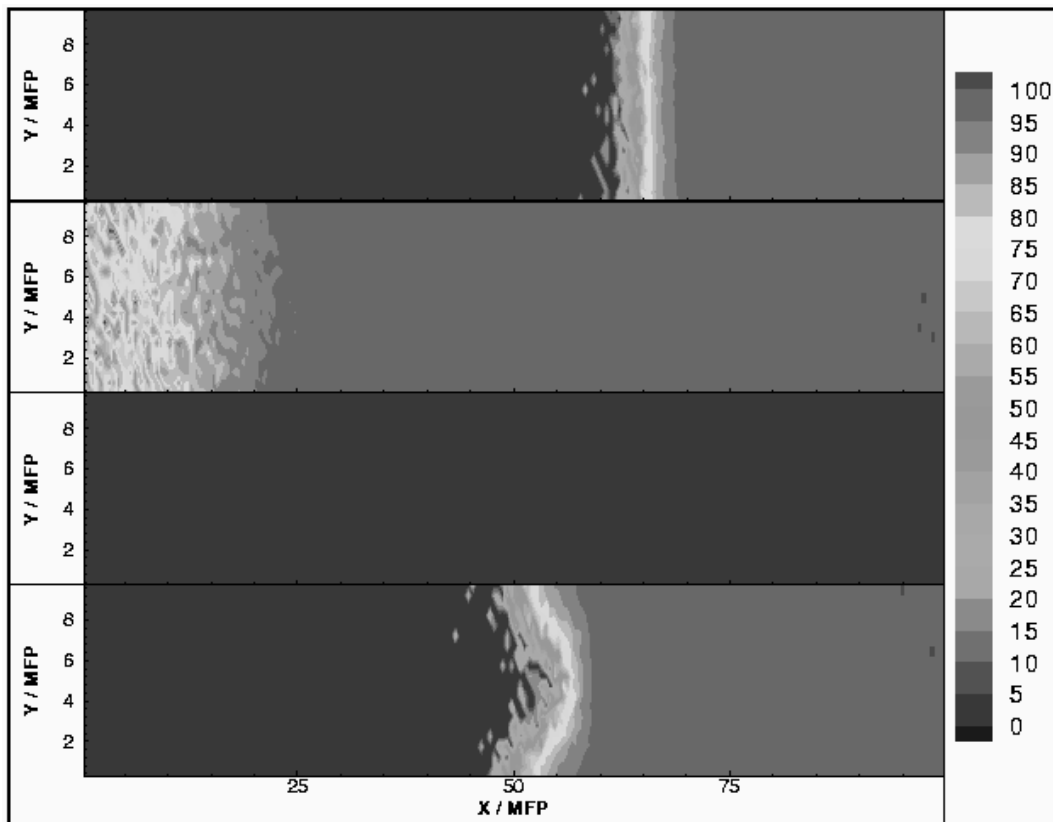


FIGURE 1. Simulation Results: Concentration of Reactant at 100 Mean Collision Times for an Idealized PDE with (from top) (a.) Specular Wall, (b.) Cold Viscous Wall, (c.) Hot Viscous Wall and (d.) Adiabatic Viscous Wall

The problems encountered when implementing real wall effects can be subdued by insulating the walls of the detonation chamber. By implementing an adiabatic viscous wall, energy initially contained in the detonation wave is allowed to remain in the detonation wave, thus allowing its propagation down the tube. Such a boundary condition subroutine was developed and works by taking the magnitude of the particle speed as a constant through a collision with the wall. The azimuth and elevation angles of the particle's trajectory must be determined as follows. If a particle comes off of the wall at elevation angle, α , with respect to the wall, the velocity component normal to the wall is $c \sin \alpha$, where c is the particle's speed throughout its collision with the wall. Since a bias in the normal velocity component is needed to reflect the tendency for particles with larger velocity components normal to the wall to collide with the wall more often than those with smaller normal velocity components, the quantity r is picked from a set of random numbers and set equal to $\sin \alpha$. The particle velocity adjacent to the wall is then $\cos \alpha = \sqrt{1 - r^2}$ and the azimuthal angle, θ , can be picked randomly, $\theta = 2\pi R_f$, where R_f is a random number between 0 and 1.

In Figure 1d., simulation results using an adiabatic viscous wall boundary condition are shown. Although the wave speed is seen to be slightly lower than in the case of the specular wall, as is apparent by noting its location at 1/2 rather than 2/3 of the tube length, a well-defined detonation wave is again formed and thrust is produced. The amount of thrust will vary with chamber cross-section, but is constant with tube length, beyond the distance over which the wave is initiated.

Figure 2 shows the thrust produced per unit exit cross-section over 10 pulse detonation cycles using the new adiabatic viscous wall boundary condition. A specific impulse of about 240 nN-sec is produced per unit cross-section. Note that this impulse occurs over a characteristic time of about 6.1 ns, corresponding to an average thrust of about 40N per square meter of cross-sectional area. The irregular spikes on the left side of the thrust time history plot shown in Figure 2 represent the first four cycles of the pulse detonation engine. Since all of the exhaust gas does not leave the tube at the end of each cycle, a few cycles are required to establish an equilibrium between the amount of exhaust gas that remains in the tube and the reactants that are injected at the start of a given cycle. Since the thrust produced is directly related to the amount of energy added to the flow during the reaction processes, the number of reactions will have a heavy bearing on how much thrust is produced in any given pulse.

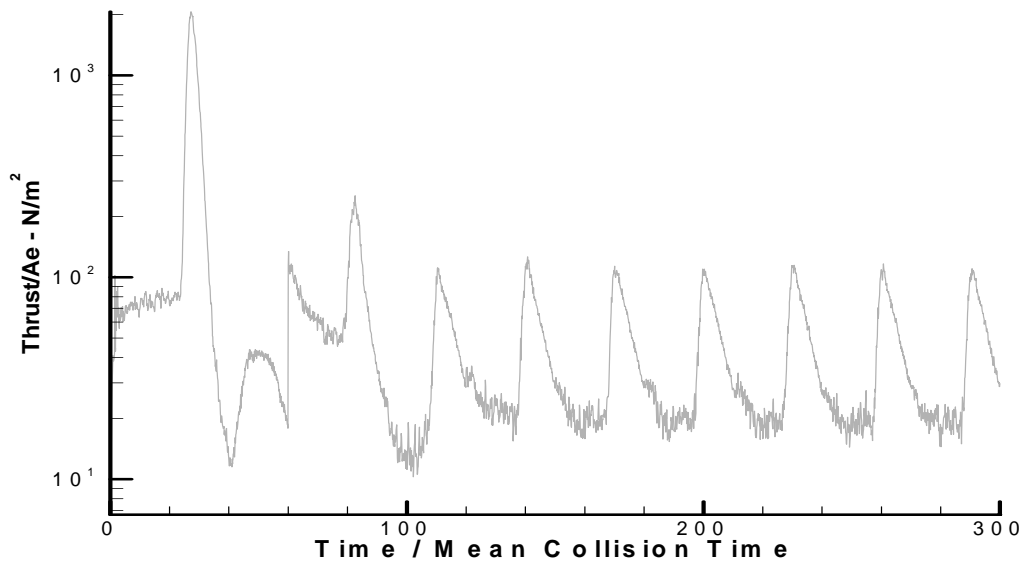


FIGURE 2. Thrust Time History for the Nano-Scale Pulse Detonation Engine Using an Adiabatic Viscous Wall Boundary Condition

PERFORMANCE TESTING

The biggest criticism of Java as a scientific computing tool has been its supposed lack of performance as compared to FORTRAN, C and C++. Specifically, Java is unable to handle multi-dimensional arrays efficiently. Early versions of this DSMC code were plagued with difficulties arising from the extensive use of these arrays. Specifically, 3-dimensional arrays of cell objects were created, as were 4-dimensional arrays of cell-sampling data. Consequently, memory requirements were impractically large, forcing the code to be rewritten. The latest version of the Java DSMC code contains only one-dimensional arrays, resulting in a code that is far more efficient than its predecessor from a memory usage standpoint

Once the modifications were completed, the code was tested and compared on a performance basis to a similar FORTRAN 90 code. The model problem for the performance testing was the Rayleigh flow problem (Stokesí First Problem), the problem of an impulsively started flat plat. The code was tested on three different computer systems, an IBM RS-6000 running AIX 4.3, a laptop computer running Microsoft Windows 2000 and a Dell Server running Red Hat 7.0. It will be apparent from the discussion to follow that the relative performance of Java compared to FORTRAN is highly dependent on machine architecture and the compiler used. The table below summarizes the matrix of hardware and software configurations tests that were run.

TABLE 1. Computer System Hardware and Software for Performance Testing

System	Processors	Compilers
IBM RS-6000 Running AIX 4.3	(4) POWER III @ 375 MHz	IBM XLF90 FORTRAN Compiler Sun Java Virtual Machine v.1.3.0
IBM Thinkpad T21 Running Microsoft Windows 2000 (v.5.0, SP1)	(1) Pentium III @ 800 MHz	Compaq Visual FORTRAN v.6.0 Sun Java Virtual Machine v.1.3.1
Dell Precision Server Running Red Hat Linux v.7.0	(4) Pentium III @ 400 MHz	Absoft FORTRAN 90 Compiler GNU GCJ v.1.0, GCC v.2.9.6 Sun Java Virtual Machine v.1.2.2

Figure 3 shows the performance of the Java code in solving the Rayleigh problem on a uniform grid of 100x1x1 cells of dimension $\lambda/2$. The simulation was run for 1,000 DSMC time steps of 0.1 mean collision times. Averages were taken across 1, 10, 100 and 1,000 ensembles as shown in the figure. Surprisingly, the Java version of the DSMC code outperformed the Fortran version on the RS-6000 by about a factor of five. Similarly, on a Laptop PC running Windows 2000, Java outperformed Fortran once again, as shown in Figure 3b. The ability of the Java code to outperform the Fortran code is an important accomplishment, especially since in both cases, the Sun Java Virtual machine was used rather than a compiler. A linear relationship is seen between the number of ensembles to be run and the necessary CPU time as expected.

A third performance test was run on a 4-processor Dell Server (running Red Hat Linux 7.0). In this case, the Fortran code was found to be superior, outperforming the interpreted Java code by approximately a factor of four and just edging out the compiled Java program. For this case, it is important to notice the performance difference between the Java Virtual Machine and the GNU GCJ compiler. Since the GCJ compiler has been available for less than one year, one would expect significant gains to be made insofar as performance optimization. With major compiler advances anticipated in the near future, Java should emerge as a very powerful language for scientific computing applications.

The next step after the performance testing on different computer architectures was the assessment of the parallel performance of the code. The code was parallelized simply by splitting the ensembles between multiple processors. The parallel performance assessment was done on the same IBM RS-6000 used earlier, using the same test case as above and performing one test over 120 ensembles and another over 1200 ensembles. The parallel performance tests were run with up to eight threads, to investigate how well Java and the operating system work together when computational resources are limited. From one to four threads, a fairly linear speed-up was observed. When the six-thread and eight-thread cases were run, a small increase in CPU time was seen due to the small amount of overhead associated with starting up another Java thread. When there are sufficient processors to perform the computations, this overhead is hardly noticeable. Generally speaking, AIX deals with the extraneous threads quite well and seamlessly swaps them between thread-states of running and sleeping. When similar experiments were run on the Windows and Linux systems, CPU time increased by as many as three orders of magnitude for the same 2,4,6 and 8-thread runs shown above, which suggests that the Java threads are not implemented well on these machines.

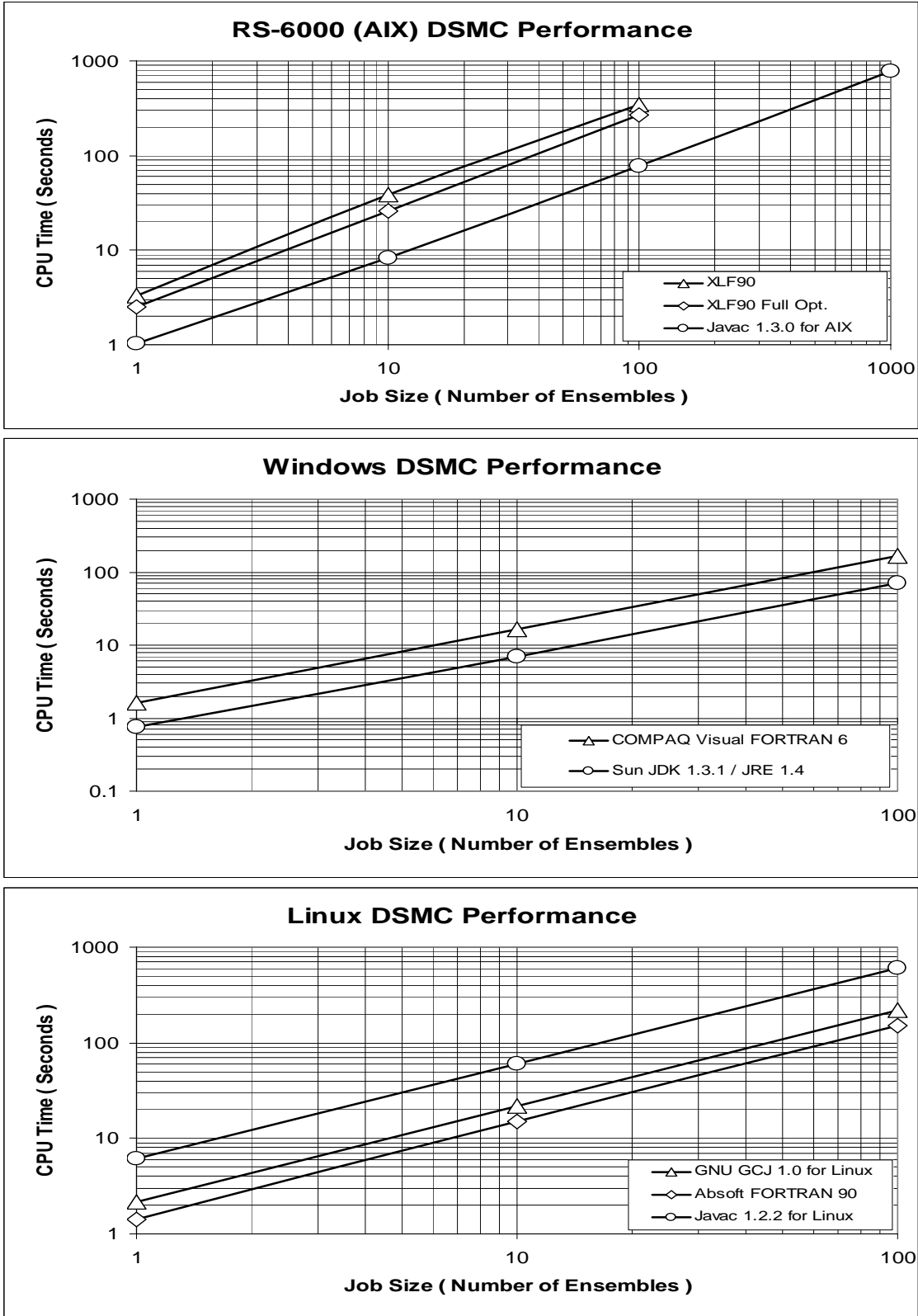


FIGURE 3. Java Performance Comparison to FORTRAN 90 in (a.) AIX, (b.) Windows and (c.) Linux

CONCLUSIONS

The idealized Pulse Detonation Engine was successfully simulated using the Direct Simulation Monte Carlo Method. The results, however, should be taken as qualitative and the value of this project must be found in the methodology by which small-scale two-dimensional detonation problems that are dominated by wall effects may now be solved. The multi-dimensional simulation of a detonation wave is a subject that will certainly be addressed in the near future. As apparent from this study and from [12], there is a definite relationship between the curvature of a detonation wave and the propagation speed. This relationship needs to be better understood and modern computer technology is finally sufficient to simulate multi-dimensional detonation systems.

Although the performance of Java as a tool for scientific computing is, in general, still a candidate for further revision, great gains have been made in developing the language as is evident by considering the performance evaluation of the code used in this report. Although the use of multidimensional arrays is convenient, there are means of getting around Java's inadequacies in dealing with them. It is also important to note that the language is very immature, especially from the scientific computing standpoint, and that these details should be worked out in the very near future.

An initiative was recently approved to run this code on a 256-processor Silicon Graphics Origin 2000. The shared memory environment that this supercomputer provides is ideal for the implementation of this code on multiple processors. Larger detonation problems may now be run in fractions of the time that it took to run the relatively small cases that were discussed in this report. Using this supercomputer, the effects of wave curvature will be studied more closely. With such a dramatic increase in computational power, it will be possible to run problems that are several orders of magnitude greater than the problems discussed herein.

REFERENCES

1. Bird, G.A., *Molecular Gas Dynamics and the Direct Simulation of Gas Flows*, Clarendon, Oxford, 1994.
2. Long, L.N. and Anderson, J.B., *The Simulation of Detonations Using a Monte Carlo Method*,[†] Rarefied Gas Dynamics Conference, Sydney, Australia, June, 2000.
3. Dunn, S.M. and Anderson, J.B., *Direct Monte Carlo Simulation of Chemical Reaction Systems: Internal Energy Transfer and Energy-Dependent Unimolecular Reaction*,[†] *J. Chem. Phys.*, Vol. 99, Number 9, November, 1993
4. Kailasanath, K., *Recent Developments in the Research on Pulse Detonation Engines*,[†] AIAA 2002-0470, 40th AIAA Aerospace Sciences Meeting & Exhibit, Reno, Nevada, January, 2002.
5. Hayashi, A.K., Mikutsu, Y. and Araki, N., *Numerical Simulation on a Pulse Detonation Engine*,[†] 17th International Colloquium on the Dynamics of Explosions and Reactive Systems, Heidelberg, Germany, July, 1999
6. Kawai, S. and Fujiwara, T., *Numerical Analysis of First and Second Cycles of Oxyhydrogen PDE*,[†] AIAA Paper 2002-0929, January, 2002
7. Benedick, W., Guirao, C., Knystautas, R. and Lee, J.H.S., *Critical Charge for the Initiation of Detonation in Gaseous Fuel-Air Mixtures*,[†] *Prog. Astro. Aero.*, Vol. 106, pp. 181-202, 1986
8. Explosion Dynamics Laboratory, The California Institute of Technology, <http://www.galcit.caltech.edu/EDL/projects,pde/pde.html>, 2001
9. Moen, I.O., Thibault, P.A., Funk, J.W., Ward, S.A., and Rude, G.M., *Detonation Length Scales for Fuel-Air Explosives*,[†] *Prog. Astro. Aero.*, Vol. 94, pp. 55-79, 1984
10. Genovesi, D.J., *A Direct Monte Carlo Simulation of a Nano-Scale Pulse Detonation Engine*,[†] Master's Thesis, The Pennsylvania State University, University Park, PA, May, 2002
11. Long, L.N. and Brentner, K.S., *Self-Scheduling Parallel Methods for Multiple Serial Codes with Application to WOPWOP*,[†] AIAA 2000-0346, January, 2000
12. Oran, E., *The Structure of Propagating Detonations: Lessons Learned from Numerical Simulations*,[†] 17th International Colloquium on the Dynamics of Explosions and Reactive Systems, Heidelberg, Germany, July, 1999