CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2000; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Implementation and Performance of a Particle In Cell Code Written in Java

S. Markidis[†], G. Lapenta[†], W.B. VanderHeyden[†], Z. Budimlić[‡]

[†] *Theoretical Division, Los Alamos National Laboratory, Los Alamos, NM 87545, USA*
[‡] *Center for High Performance Software Research, Rice University, Houston, TX 77005, USA*

KEY WORDS: PIC Method, OO programming, High Performance Computing, Java

SUMMARY

Plasma simulation is an important example of a high performance computing application where computer science issues are of great relevance. In a plasma, each particle, electron or ion, interacts with the external fields and with other particles in ways that can be readily and effectively emulated using object oriented programming. However, the great cost of plasma simulations has traditionally discouraged object oriented implementations due to their perceived inferior performance compared with classic procedural FORTRAN or C. In the present paper, we revisit this issue. We have developed a Java particle in cell code for plasma simulation, called *Parsek*. The paper considers different choices for the object orientation and tests in practice their performance. We find that coarse grained object orientation is faster and practically immune from any degradation compared with a standard procedural implementation (with static classes). The loss in performance for a fine grained object orientation is a factor of about 50%, which can be almost completely eliminated using advanced Java compilation techniques. The Java code Parsek provides also an interesting realistic application of high performance computing to compare the performance of Java with FORTRAN. We have conducted a series of tests considering various Java implementations and various FORTRAN implementations. We have also considered different computer architectures and different JVMs and FORTRAN compilers. The conclusion is that with Parsek, object oriented Java can reach CPU speed performances more or less comparable with procedural FORTRAN. This conclusion is remarkable and it is in agreement with the most recent benchmarks but is at variance with widely held misconceptions about the alleged slowness of Java.

1. INTRODUCTION

Simulation of the behavior of plasmas is an ideal candidate for object-oriented techniques. A plasma is, by definition, a collection of objects (the plasma particles, electrons and ions) with several properties and with well defined functional interactions with other plasma objects.

Received
Revised

Computationally, it is quite natural to think of plasmas in terms of objects with data that each object must carry and methods that each object should possess. Moreover, the computational schemes that have been used to simulate plasmas have also the structure of objects. For example superparticles and grid nodes of the classic Particle-in-Cell (PIC) simulation [1, 2] can be directly regarded as objects. Finally, simulation of plasmas must include different approaches, branches of physics, and levels of expertise. The plasma species themselves, for instance, can be treated either in a kinetic or fluid dynamic approach. The electromagnetic fields, however, use neither of these approaches but require the solution of full or approximate Maxwell equations. Chemical interactions among the charged plasma species require different expertise - complex chemistry - as in plasma processing devices for semiconductor production [3]. Sometimes molecular dynamics or solid mechanics models are required. Creating a simulation as a series of objects allows for seamless integration of the appropriate expertise.

Previous successful attempts to create an object oriented plasma simulation [4, 5] have had limited application either because the language used (FORTRAN 90 [4]) did not support objects well or because the language (C++ [5, 6]) imposed a high implicit cost in implementing changes to the simulation. Here we use Java to create a flexible computationally rapid implementation of PIC for plasma simulations. Java's object-oriented programming allows the abstraction of common physical concepts and the development of reusable class libraries. Java byte code is portable across multiple platforms, and Java's multithreading allows parallel computation critical for high performance plasma simulations. Furthermore Java allows for an effective project management. Previous studies [7] have shown that Java programming results in fewer bugs and faster code development.

Previous experience with Java has shown that a complete object oriented design generally results in slow execution. Typically, the penalty is a factor of ten or more [8]. If the objects are large, including arrays of data ("coarse grained object oriented"), the execution time penalty may be smaller. For plasma simulations, species level objects instead of particle level objects may increase computation speed.

We have initiated a long term project to develop a complete software package for plasma simulation fully written in object oriented Java. We called the project: Parsek. In the present study, we compare the execution speed and the solution correctness of different versions of Parsek all written in Java but with different implementations: FORTRAN-style (procedural Java with static classes), coarse grained style and fine grained style. FORTRAN style and coarse grained style execute at approximately the same speed. The fine grain style is only approximately 1.5 times slower when compiled with traditional compilers, and competitive with coarse grained style and FORTRAN style when compiled with an advanced optimizing compiler [9].

We have also compared the Java versions of Parsek with different implementations written in FORTRAN 90. A widely held belief in the high performance computing community is that Java can be an order of magnitude slower than FORTRAN [10]. Recent rigorous benchmarks conducted at the National Institute for Standard and Technology (NIST) and based on a suite of widely used numerical algorithms has proved otherwise: Java and FORTRAN are now basically equally efficient, achieving approximately the same speed measured in floating operations per second (FLOPS) [11]. Our study indeed confirms such conclusion: different implementations of Parsek written in Java and in FORTRAN can vary their execution time somewhat, but on average, FORTRAN and Java run at comparable speed.

The dramatically improved execution times of our Java program in comparison to previous comparisons of Java and FORTRAN [8, 10] appear to be caused by the improved compilation environment (Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.0-b92) Java HotSpot(TM) Client VM (build 1.4.0-b92, mixed mode)).

2. SCIENTIFIC CHALLENGES

The simulation of systems where plasmas are present requires the description not only of the scale of interest but also of the smaller scales that affect the physics of the systems under consideration. For instance, simulation of coronal mass ejection from the Sun requires the description of large scale processes using a magnetohydrodynamic (MHD) model [12]. However, the MHD models require to include models of dissipation processes that develop at microscopic scales. The calculation of dissipations requires more accurate microscopic kinetic models, beyond the fluid approach. At small scales dissipations are present not only as interparticle collisions but also through electromagnetic interactions of ions and electrons at the microscopic scales. A self-consistent description of astrophysical systems must be performed at the kinetic level using the Boltzmann equation for ions and electrons, the Maxwell equations for the electromagnetic fields and the Newton (or Einstein) equation for the gravitational field. However the cost of such direct approach would be prohibitive if attempted using the most common explicit methods currently in use. The standard approach is to represent the systems with reduced models such as the Hybrid, resistive MHD, Hall MHD or two fluid model, where some or all species are approximated in the fluid limit [13]. In all reduced models, ad hoc assumptions of the kinetic behaviour are made, most commonly in the form of prescriptions for the higher order moments of the distribution (e.g., the pressure tensor) and for the dissipation processes (e.g., anomalous resistivity) [13].

We follow a bolder approach [14, 15]: we adhere to the exact kinetic model with all the correct microscopic physics. To be able to bring such approach all the way to the large scales of interest we use two powerful techniques that can make the numerical simulation manageable within the existing computing resources: object orientation and implicit formulation. The implicit formulation is described elsewhere [16] and its description is beyond the scope of the present paper. Here we use a simpler explicit PIC algorithm and focus only on the issue of object orientation of a plasma simulation code which is described next. Below, we report a simple version of the Particle In Cell method. The scheme considered here is a full fledged plasma simulation method currently being widely used in the plasma physics community [1, 2]. Our goal here in not simply to use an artificially simple benchmark to test Java performances but to test Java in a realistic application.

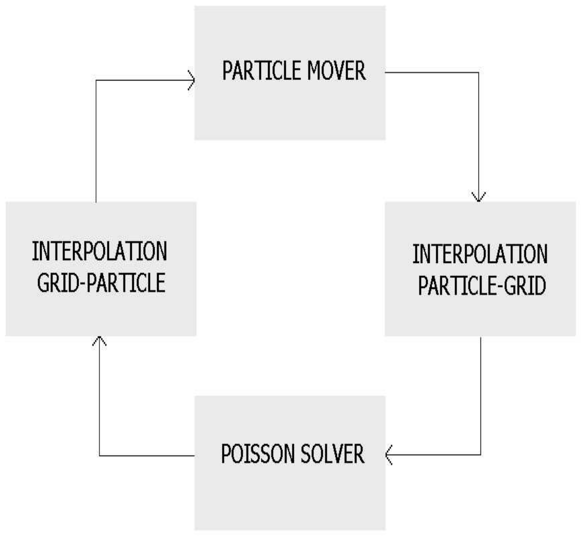

Figure 1. An Explicit Electrostatic PIC Algorithm.

2.1. Skeleton Particle in Cell Algorithm

Our simplified algorithm consists of three parts: the interpolation scheme, the Poisson solver and the particle mover, as shown in Fig. 1. A complete description of the PIC algorithm can be found in textbooks [1, 2]

Interpolation particle-grid

The density on the grid is calculated from the particles through the interpolation scheme defined by

$$\rho_i = \sum_p \frac{q_p}{\Delta x} W(x_i - x_p) \tag{1}$$

where i and p label grid nodes and particles, respectively. Δx is the space step while q_p is the particle charge. The classic Cloud-In-Cell (CIC) method [1, 2] is used for the interpolation functions:

$$W(x_i - x_p) = b_1(\frac{x_i - x_p}{\Delta x}) \tag{2}$$

where the b_1 is the first order b-spline function [2].

Field Solver

The Poisson equation for the electric potential Φ is:

$$\frac{d^2\Phi}{dx^2} = -\frac{\rho}{\epsilon_0} \tag{3}$$

where ϵ_0 is the dielectric constant. Equation (3) is solved using a finite difference scheme. Then the electric field E on the grid can be calculated by using the central difference discretization and solving the resulting linear system with Gaussian elimination.

Interpolation grid-particle

Given the electric field on the grid, the electric field on each particle can be calculated using the same CIC interpolation scheme:

$$E_p = \sum_i E_i W(x_i - x_p) \tag{4}$$

Particle mover

Particles are moved solving the Newton equations of motion for the particle position x_p and velocity v_p :

$$\frac{dv_p}{dt} = \frac{q_p}{m_p} E(x_p) \tag{5}$$

$$\frac{dx_p}{dt} = v_p \tag{6}$$

where m_p is the particle mass. Equations (5, 6) are discretized with the leap frog finite difference scheme [1, 2].

3. OBJECT ORIENTED IMPLEMENTATION

The biggest problem for any advanced plasma simulation code is to organize the complexity. Because plasma physics simulations are becoming more complex and because more physicists become involved in the writing of software, we need more sophisticated and easier development techniques. With an object-oriented framework, the computational physicist tries to organize the physical problem into objects which control the complexity of the simulation.

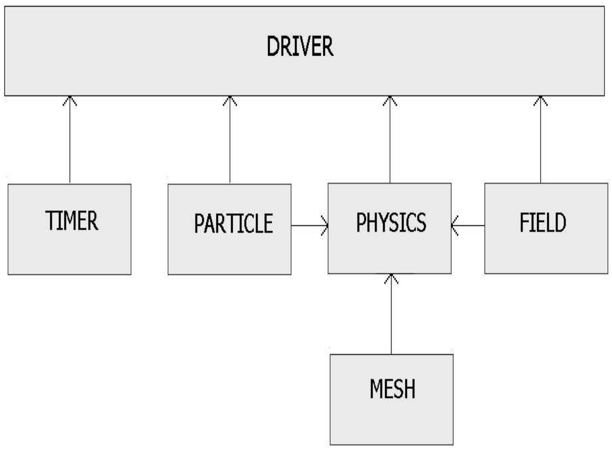

Figure 2. Parsek Framework.

In designing Parsek, we have tried to follow two guiding principles. First, we have tried to use a full object-oriented programming from a physical point of view. Object orientation gives an elegant software design and results in a code that is easy to read and can be more effectively used by physicists who are less proficient in computer science issues.

Second, we have written the code to be as generic as possible so that the computer programmer can plug plasma physics into the proper program locations and develop a new code to study different plasma phenomena. So the programmer can extend the functionality of the code by adding models and algorithms of various level of complexity.

The algorithm discussed and tested in the present work is a skeleton version of a complete plasma simulation code. We use a PIC scheme that includes all the most important steps present in a complete code. The algorithm summarized in Sect. 2.1 follows the trajectories of a number of particles in force fields which are calculated self-consistently from charge, current and pressure densities created by the particles. Each time step in a PIC code consists of two major steps: the *particle mover* to update the particle positions and calculate the new charge and current densities, and the *field solver* to update the surrounding fields. Since particles can be located anywhere within the simulation domain but the surrounding fields are defined only on discrete grid points, the *particle mover* uses two interpolation steps to link the particle positions and the fields: a step to interpolate fields from the grid points to the particle positions and a step to interpolate the charge of each particle to grid points. The Parsek architecture is summarized in Fig. 2. The complete code listing is too long to be reported here, as is to be expected for a realistic simulation code that can study realistic physics problems. Parsek is composed of six separate classes:

- **Particle Object**
  The *Particle* class describes an individual plasma particle, like an electron or ion, including its position and velocity. Basically *Particle* is organized as:

  ```
  public class Particle {
       private double Position;
       private double Velocity;
       private double ElectricFieldOnParticle;
       ...
  }
  ```

- **Field Object**
  The *Field* class represents the electromagnetic field and its sources for a given point of the mesh. The *Field* object includes the charge density and the electric field for each grid node. It is written as follows:

  ```
  public class Field {
      private double ChargeDensity;
      private double ElectricField;
      ...
  }
  ```

- **Mesh Object**
  The *Mesh* class contains several methods that describe mesh elements and boundary nodes. Furthermore, it provides methods to calculate discrete differential operators and to interpolate a discrete vector field onto a specified location in the mesh.
- **Physics Object**
  The *Physics* class handles the particle *mover phase*, where the new particle position and velocity are determined by Newton's law, and the *field solve phase*, where the fields are updated solving Maxwell's equations.
- **Driver Object**
  The *Driver* class describes the methods that handle the whole simulation. After initializing the arrays of particle and field objects with

  ```
  Particle[] myParticle = new Particle[NumberOfParticles];
  Field[] myField = new Field[NumOfGridPoints];
  ```

  the initial conditions for the particle velocities and positions are set. Once constructed, the simulation is advanced in discrete units of time. Fields are calculated from the sources, including the appropriate boundary conditions. At this point, the explicit method requires to solve a linear system to determine the new electric field. Next, the forces on particles are calculated by interpolating the fields to the particle positions. The forces are used to update the particle velocity, and subsequently the particle position. These procedures are repeated for each incremental time step.

- **Timer Object**
  Finally, the *Timer* class calculates the time performance of the code.

4. ALTERNATIVE IMPLEMENTATIONS OF PARSEK

The object oriented implementation of Parsek described above uses a fine grained approach. The objects are chosen to correspond to the smallest units in the physical system under consideration: the particles and the mesh points. Alternative approaches are possible.

Previous studies have led the high performance computing community to reach two widely held beliefs [8, 10].

First, programs written in Java are believed to be an order of magnitude, or more, slower than corresponding programs written in C or FORTRAN [10]. To ascertain this point we have developed various FORTRAN and Java versions to compare their relative speed.

Second, fine grained object orientation, either in C++ or in Java, is believed to be much slower than coarse grained object orientation [8]. Fine grained object orientation can be loosely defined as the choice to define objects at the smallest scale of interest in the problem being considered. For plasma simulation this corresponds to the choice outlined in the previous section where the objects were chosen as single particles and single mesh points. The crucial feature of fine object orientation is that the objects are small and large arrays of them are required. The additional cost of handling arrays of objects is believed to result in a great penalty in terms of computing efficiency. Coarse grained object orientation, instead, defines broader objects that include larger units of the system under consideration. For plasma simulations this corresponds to choosing objects composed by the whole grid or by whole populations of particles (such as all ions or all electrons). The crucial feature of coarse object orientation is that all relevant arrays are wrapped inside the objects and no arrays of objects are required. Previous studies [8] have reported penalties of one order of magnitude when fine grained object orientation is compared with coarse grained object orientation and only the traditional compiler techniques are used. To test this issue we have developed different versions of Parsek all written in Java but using different object orientation styles.

The two beliefs described above are often based on evidence obtained some years ago when the Java virtual machines and compilers were still in their infancy. Furthermore, often such conclusions were reached using simple methods not applied to any scientific problem. More recently, extensive benchmarks of Java using a suite of standard mathematical problems has shown that contrary to the commonly held beliefs, Java is almost on par with FORTRAN [11].

Here we intend to conduct all tests with the most modern compilers on the most modern computer architectures. And we will conduct all tests for a real problem of plasma simulation where the final answer is a significative plasma physics result. While most of the previous performance studies were conducted on benchmark problems, we will base our study on a realistic plasma physics simulation tool.

Below, we put the two beliefs described above to test using several alternative versions of Parsek both in Java and in FORTRAN 90. All versions are equivalent from the algorithmic point of view but are radically different in the choice of software architecture and programming language. Below we describe the various versions.

4.1. Coarse Grained Object Oriented Parsek

Two approaches to object orientation are possible: a "coarse grained object-oriented" (referred to as LOO) and a "pure object-oriented" (referred to as OO) programming style. We have described the OO design in the section above. With a LOO technique the arrays that describe particles and fields are wrapped in two objects that represent the whole particle population and electrostatic field states. The coarse grained object-oriented Parsek is composed by 5 separate classes:

- **Particles Object**
  The Particles class in the LOO framework acts as container to store the characteristic data for N individual particulate elements. Each individual particle has several attributes, such as position and velocity. In the code, examples are:

  ```
  private double[] Position = new double[NumberOfParticles];
  private double[] Velocity = new double[NumberOfParticles];
  ```

  In a LOO code arrays are wrapped in a single object, and no array of objects is used. In the fully OO PIC code, instead, objects were single particles and arrays of objects were used. Moreover the *Particles* object contains methods to move and accelerate the particles, and to check if the particles are leaving the boundaries. Unlike the case of the OO PIC code, in a LOO PIC code there is a direct interaction between the *Particles* and *Mesh* objects.
- **Fields Object**
  A Fields object represents a discretization of a continuous field quantity over an underlying mesh. Internally, Fields data is stored essentially as an array, containing charge density, potential, and electric field values on the grid. In Java, it is written as follows:

  ```
  private double[] ElectricField =  new double[NumOfGridPoints];
  private double[] Potential = new double[NumOfGridPoints];
  private double[] ChargeDensity = new double[NumOfGridPoints];
  ```

  The *Field Solver* is a method of this class.
- **Mesh Object**
  It contains the informations about the grid and methods to calculate the interpolation functions.

- **Driver Object**
  It coordinates the other objects and controls the progress of the computational cycle.
- **Timer Object**
  The *Timer* object calculates the timing performance of the code.

4.2. FORTRAN Style Parsek

Although Java is a full fledged object oriented language, old fashioned procedural programming remains possible using static classes [18]. We have developed an additional Java version of Parsek that uses a "FORTRAN style" (FS) procedural program. All methods are static, arrays are passed directly as arguments and the data is accessed directly. The FORTRAN style code is procedural in only one class. It includes the usual *Particle mover*, *Field Solver*, and *Interpolation* stages.

4.3. FORTRAN 90 Parsek

To compare Java and FORTRAN 90 performances we wrote two additional versions of Parsek in FORTRAN 90. We have chosen to use modern FORTRAN90 features, including types, modules and array notation. Two versions have been written in FORTRAN 90: one with coarse grained types and one with fine grained types.

4.3.1. Fine grained types

The fine grained data is stored as an array of elements of a defined type. The *Particle* type describes an individual plasma particle, like an electron or ion, including its position and velocity. *Particle* is organized as:

```
TYPE Particle
    DOUBLE PRECISION  :: Position
    DOUBLE PRECISION  :: Velocity
    DOUBLE PRECISION  :: ElectricFieldOnParticle
END TYPE Particle
```

The *Field* type represents the electromagnetic field and its sources for a given point of the mesh. The *Field* type includes the charge density, the current density, the electric field for each grid node. *Field* is written as follows:

```
TYPE Field
    DOUBLE PRECISION :: ChargeDensity
    DOUBLE PRECISION :: Potential
    DOUBLE PRECISION :: ElectricField
END TYPE Field
```

4.3.2. Coarse grained types

The coarse grained data is stored as a defined type that includes arrays to accomodate particle and field data. The *Particles* type describes an entire species of particles (composed of NumberParticles individual particles), like all electrons or all ions, including their position, velocity, mass, charge and the electric field acting on them. *Particles* is organized as:

```
TYPE Particles
    DOUBLE PRECISION,dimension(0 : NumberParticles-1)  :: Position
    DOUBLE PRECISION,dimension(0 : NumberParticles-1)  :: Velocity
    DOUBLE PRECISION,dimension(0 : NumberParticles-1)  :: Charge
    DOUBLE PRECISION,dimension(0 : NumberParticles-1)  :: Mass
    DOUBLE PRECISION,dimension(0 : NumberParticles-1)  :: ElectricFieldOnParticle
END TYPE Particles
```

The *Fields* type represents the electric field and its sources for the whole mesh (composed of NumOfGridPoints points). The *Fields* type includes the charge density, the current density, the electric field for each grid node. It is written as follows:

```
TYPE Fields
    DOUBLE PRECISION, dimension(0 : NumOfGridPoints-1) :: ChargeDensity
    DOUBLE PRECISION, dimension(0 : NumOfGridPoints-1) :: Potential
    DOUBLE PRECISION, dimension(0 : NumOfGridPoints-1) :: ElectricField
END TYPE Fields
```

5. PARSEK AS BENCHMARK

The goal of our work is two-fold. First, we intend to develop a pure object-oriented simulation code to study space and astrophysical plasmas. Second, we want to develop a benchmark that closely reflects the "real world" scientific computation. In the present paper, we consider only a simplified version of the complete implicit PIC algorithm that we will ultimately use [16]. The simplified algorithm is reported in Sect. 2.1 and includes all the relevant steps used in the great majority of existing PIC codes and provides a realistic benchmark for Java.

Using Parsek as a benchmark we conclude that a fine grained object-oriented design is penalizing in performance but only by a reasonable margin. Furthermore, we reach the somewhat surprising conclusion that Java has already become competitive with FORTRAN 90 for state of the art scientific computing. Below we describe in detail the results of our testing campaign.

5.1. Testing Environment

Table I presents the platforms used for measuring the performance. The Java environment is reported in Table I for each platform. The machines were relatively unloaded during each simulation, and several runs were made at each test condition with the average time recorded.

Table I. Platform used to test Parsek

Model	Sun Blade 2000	Dell Precision 520	Dell Latitude C840	Dell Latitude C840
Processor	UltraSPARC-III+	Xeon	Pentium 4	Pentium 4
Processor Speed	1.2 GHz	1.9 GHz	1.6 GHz	2.0 GHz
Memory	512MB	4GB	512MB	1GB
Operating system	SunOS 5.8	RedHat Linux 8	Windows 2000	Windows XP
SUN Java build	1.4.1-02-b06	1.4.1-b21	1.4.2-b28	1.4.1-b21

5.2. Test Problem

We study one-dimensional plasma dynamics in an uniform grid. In these simulations two equal Maxwellian streams of electrons flow through each other, and the fields are solved in the electrostatic limit. A motionless background of ions provides charge neutrality. This plasma physics problem is known as the ”two stream instability” and it has been studied thoroughly [1, 2]. The system extends between $x = 0$ and $x = 2\pi$, with periodic boundary conditions. The ions are motionless, while the electrons follow trajectories imposed by the interactions with the other charged particles. We have performed simulations with different number of particles in a grid of 64 nodes.

Figure 3 shows the time evolution of the total energy of the physical system, as sum of kinetic and potential energies. Figure 4 presents the phase space plot after 1000 time steps. Every point of this plot identifies a particle: the projection on the x-axis is its position, while the other coordinate is its velocity. Figures 3, and 4 show the results obtained with the three different Java implementations (OO, LOO and FS) and prove that the three different Parsek implementations reach the same results and that they perform equivalent operations. Note that we used the same seed to generate the same pseudorandom series.

5.3. Java Performance

Table II and Figures 5 and 6 compare the execution time for the different implementations of Parsek. All tests are performed on the SUN platform. In Table II the second, third and fourth columns show the performance of Parsek, developed in Java with different object orientation techniques. Each row presents the time performance in milliseconds, for simulations with a different number of particles showed in the first column. Figure 5 illustrates graphically these results. Figure 6 describes the dependency of the timing performance on the number of particles. Clearly, the timing shows the relative slowness of the fine grained object-oriented programming solution. While FS and LOO show a difference in performance of only a few percent, the OO Parsek is on average 1.5 times slower than the others. However, it must be stressed that, in our PIC code, the fine grained object-orientation is not as penalizing as reported in previous studies [8]. Budimlić and Kennedy [9] wrote an object-oriented set of LINPACK classes that they called OwlPack (Objects Within Linear algebra PACKage) [9]

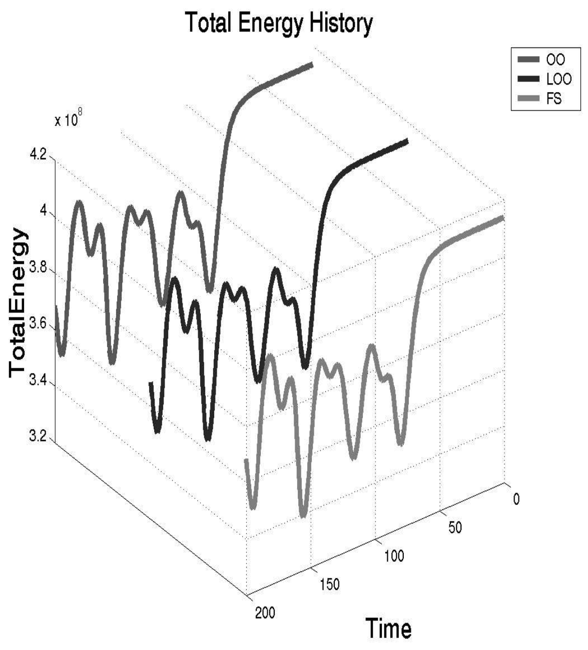

Figure 3. History of Total Energy for the three implementations. The codes show the same evolution.

and tested the different programming techniques. Based on OwlPack, the conclusion was that a OO program can be up to twenty times slower than the FORTRAN-style one.

Recently ”The Center For High Performance Software” of the Rice University has developed JaMake [9], a Java compilation environment that uses advanced program analysis and transformation techniques. JaMake is able to improve the performance of a fine grained object oriented program to bring it almost to the same speed as a procedural or coarse grained object oriented program. Although the JaMake package is still under development, we have succeded in testing the performance of the OO version of Parsek when compiled using JaMake. The last column of Table II shows the timing performance of the OO version of Parsek compiled with JaMake. The results clearly show the elimination of the additional costs of fine grained object orientation.

5.4. Java VS FORTRAN 90

Table III and Figure 7 compare the three Java implementations of Parsek with the two implementations in FORTRAN 90. Different platforms and operating systems are considered. All tests are conducted using 50,000 particles. In Table III the six last columns show the

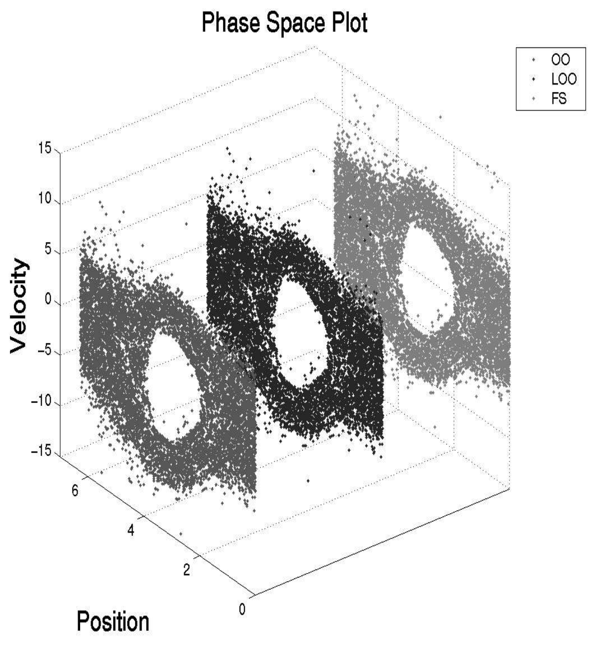

Figure 4. Phase Space plot after 1000 time steps ($\omega_{pe}t = 1$) for the three implementations. The results of the codes are identical since we used the same seed to generate random numbers and the programs perform the same operations.

Table II. Execution times, showing the almost 1.5 times speedup of FS and LOO over the pure OO. Tests on the SUN platform shown in Table I.

num.particles	OO(ms)	LOO(ms)	FS(ms)	OO-JaMake (ms)
1000	210	128	118	167
2000	413	233	233	281
5000	1032	551	605	603
10000	1999	1063	1224	1206
20000	4012	2143	2419	2517
50000	10273	5799	6164	6258

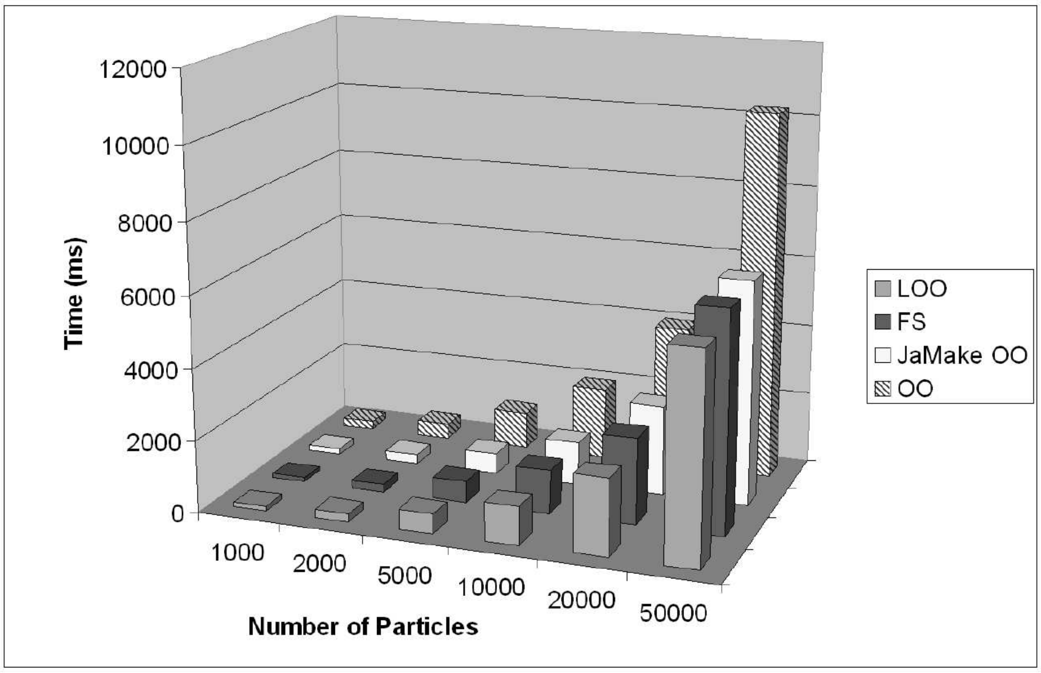

Figure 5. Execution times for FS, LOO, OO Parsek codes in Java. The performance data is reported in Table II.

performance of Parsek, developed in FORTRAN 90 and in Java with different programming techniques. The last column considers the fine grained OO version in Java compiled with JaMake. Each row presents the time performance in milliseconds, for simulations running under different operating system showed in the first column. The version of Java used is listed in Table I for each platform. For the two FORTRAN implementations, we use the Lahey FORTRAN 95 compiler (version 5.7 for the Windows platform and version 6.1 for the Linux platform) with maximum optimization on the Linux and Windows platforms. The Sun FORTRAN 90 version 6.2 with the compiler option -O3 is used for the SUN platform. Figure 7 illustrates graphically these results.

Clearly, on the SUN platform, FORTRAN and Java performances are comparable, with some Java implementations even outrunning both FORTRAN implementations. Conversely, on the INTEL platforms, the coarse grained FORTRAN version remains about a factor of two faster than the fastest Java, but the fine grained FORTRAN version is actually slower than some Java implementations. The direct comparison between Java and FORTRAN requires further comments.

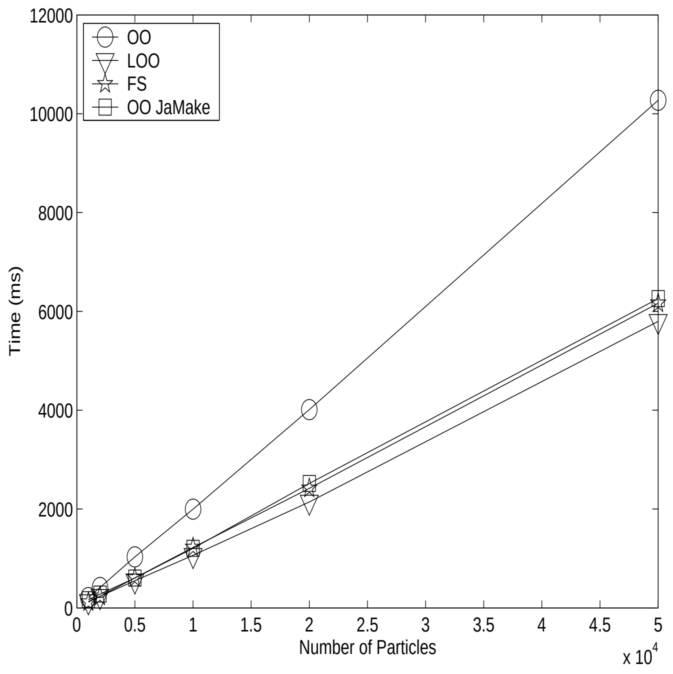

Figure 6. Execution times: increasing the number of particles, the execution time increases linearly. The performance data is reported in Table II.

Table III. Execution times for a simulation with 50,000 particles under different operating system shown in Table I

O.S.	F90 fine(ms)	F90 coarse (ms)	JavaFS(ms)	JavaLOO(ms)	JavaOO(ms)	JavaOO-JaMak
Windows2000	3765	1680	3255	3956	5800	3625
Linux	2742	1130	2874	3420	3960	4250
SUN	7430	6485	6164	5799	10273	5386

First, on the Windows 2000 platform we tested also the Compaq FORTRAN compiler that resulted in considerably slower execution. On the Linux platform we also tested the ABSOFT compiler version 7, which was also slower but by a smaller margin.

Second, the two FORTRAN implementations perform significantly differently on the two INTEL platforms. Coarse grained typing results in a improved handling of the cache since operations conducted on an array of quantities (such as the particle positions) are closer in memory and are loaded in the cache all together in a block.

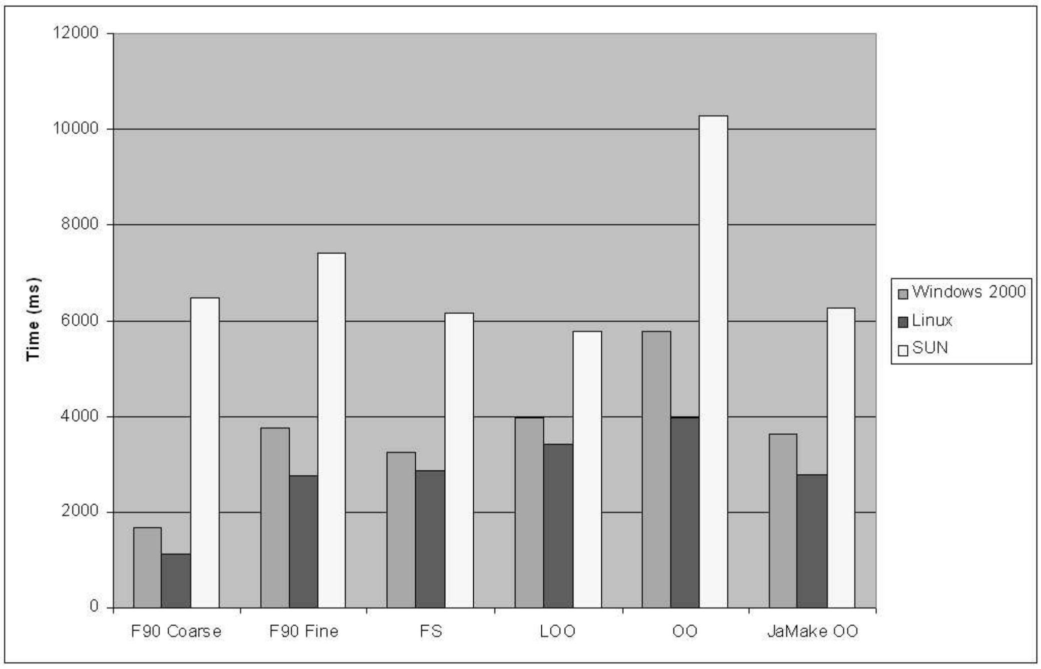

Figure 7. Execution times for FS, LOO, OO Java implementations and the two Fortran90 implementations. The performance data is reported in Table III.

Third, we have repeated the tests above with a different number of particles, reaching virtually identical conclusions.

6. CONCLUSIONS

We have examined three different choices for the object-orientation of a particle in cell code and we have studied their performances in Java. The coarse grained object-oriented version and the procedural (FORTRAN style) version perform at approximately the same speed while the fine grained object oriented version is only 1.5 times slower using a traditional compiler, and on par with coarse grained implementation when using JaMake. Moreover, we completed a series of tests comparing the Java versions with two FORTRAN 90 procedural implementations. The FORTRAN versions vary in their performance according to the specific compiler and to the choice between coarse and fine typing. But on average, FORTRAN and Java implementations run at more or less comparable speed.

The conclusions reached here are a remarkable confirmation of the applicability of Java to scientific computing. Previous studies had reported much less favorable results where fine

Table IV. Comparison in speed between SUN J2SDK1.4.1, SUN JDK1.0.2 and IBM WSDK version 5 on the Windows XP platform described in Table I

Java Version	OO(ms)	LOO(ms)	FS(ms)
SUN J2SDK 1.4.1	5357	2668	2674
SUN JDK 1.0.2	87512	49572	55409
IBM WSDK v.5	6909	2954	2624

grained object orietation was observed to be about a factor of ten slower and Java was observed to underperform FORTRAN by a wide margin [8]. We believe the improved speed of Java when compared with FORTRAN to be due to three reasons. First the Java codes we developed do not employ significantly the *Garbage collector*, since the objects are created at the beginning of the run and never destroyed. Second, the recently developed advanced compiler techniques employed in the JaMake [9] compiler can remove the overhead associated with using fine grained objects in scientific computation, as evidenced with results we have reported in this paper. Third, as observed in previous studies [11], JVMs are being improved at a fast pace by the major software companies. To prove this last point, we performed tests using the OO version of Parsek and using different JVMs. Table IV compares the timing performances, obtained with a Sun JDK 1.0.2 and a Sun J2SDK 1.4.1, running a simulation of 50,000 particles. Clearly, the new JVM is an order of magnitude faster than the old one. This factor of ten is indeed what is required to explain the improvement of Java vs FORTRAN that our tests have shown when compared with older studies [9, 10]. For completeness, Table IV also considers another Java environment from another producer (IBM), obtaining only slightly different results.

We believe that our tests show that Java even in its current version and without any special features or any additional tools can be effectively used for high performance scientific computing. Future developments, such as the JaMake compilation environment [9], will improve the performance even further, making Java a very compelling candidate for the next generation of high performance scientific computing applications.

ACKNOWLEDGEMENTS

One of the authors (S.M.) would like to thank Dr. Frederic R. Fairfield, President and CEO of the Fairfield Enterprises. This work is funded by the National Nuclear Security Agency (NNSA) and by NASA under the Sun Earth Connection Theory Program (SECTP). Z.B.'s work on this paper is supported in part by Contract No. 74837-001-0349 from the Regents of University of California (Los Alamos National Laboratory) to William Marsh Rice University.

REFERENCES

1. Birdsall C.K., Langdon A.B. *Plasma Physics via Computer Simulation.* McGraw-Hill, 1985.
2. Hockney R.W., Eastwood J. W.*Computer simulation using particles.* Adam Hilger, 1988.
3. Lapenta G., Brackbill J.U. Simulation of dust particle dynamics for electrode design in plasma discharge. *Plasma Sources Science & Technology* 1997;**6**:61-69.
4. Norton C.D., Decyk V., Slottow J. Applying FORTRAN 90 and object-oriented techniques to scientific application. *Lecture Notes in Computer Science* 1998; **1543**:462-463.
5. Verboncoeur J.P., Langdon A.B., Gladd N.T. An Object-oriented electromagnetic PIC code. *Computer Physics Communications* 1995; **87**:199-211.
6. Reynders J.V.W., Forslund D.W., Hinker P.J., Tholburn M., Kilman D.G., Humphrey W.F., OOPS: an object oriented particle simulation class library for distributed architectures. . *Computer Physics Communications* 1995; **87**:212-224.
7. Phipps, G. Comparing Observed Bug and Productivity Rates for Java and C++. *Software: Practice and Experience* 1999; **29**: 345-358.
8. Budimlić Z., Kennedy K., Piper J. The Cost of Being Object-Oriented: A Preliminary Study. *Scientific Computing* 1999; **7(2)**: 87-95.
9. Budimlić Z., Kennedy K. JaMake: A Java Compiler Environment. *Third International Conference on Large Scale Scientific Computing* 2001.
10. Lu Q.M., Cai D.S., Implementation of parallel plasma particle-in-cell codes on PC cluster, *Computer Phys. Comm.* 2001; **135**: 93-104.
11. Pozo, R., Java for High Performance Computing, *Joint ACM Java Grande - ISCOPE 2002 Conference*, Seattle, Washington, November 3-5, 2002
12. Priest E.R., Forbes T. *Magnetic Reconnection.* Cambridge University Press, 1999.
13. Lipatov A.S. *The Hybrid Multiscale Simulation Technology.* Springer-Verlag, 2001.
14. Brackbill J.U., I.B. Cohen *Multiple time scales.* Academic Press, 1985.
15. Lapenta, G., Brackbill, J.U. *Implicit Adaptive Grid Plasma Simulation*, 5th International School/Symposium for Space Simulation, Kyoto, Japan, March 13-19, 1997.
16. Ricci P., Lapenta G., Brackbill J.U. A Simplified Implicit Maxwell Solver. *J. Comput. Phys.* 2002; **183**:117-141.
17. VanderHeyden W.B., Dendy E.D.,Padial-Collins N.T 2001.CartaBlanca- A Pure-Java, Component-based Systems Simulation Tool for Coupled Non-linear Physics on Unstructrued Grids. *JOINT ACM Java Grande - ISCOPE 2001 Conference, Stanford, California* 2001.
18. Davies R., *Java For Scientists and Engineers* Addison-Wesley: University of Cambridge, 1999.