Proc. Seventh SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, February, 1995, SIAM, Philadelphia, 1995, pp. 207-212

Parallel Object Oriented Implementation of a 2D Bounded Electrostatic Plasma PIC Simulation*

Charles D. Norton† Boleslaw K. Szymanski† Viktor K. Decyk‡

Abstract

We discuss the software development issues involved in designing parallel programs using object oriented techniques. Simulations involving 1D and 2D Particle In Cell plasma codes illustrate how C++ programs can effectively describe complex simulations while performing with reasonable efficiency when compared to the equivalent Fortran programs. The scalable object oriented modeling techniques closely match the physical view of the problem, thus supporting modifiability and portability of the code. Selection of a parallel programming paradigm must consider the important factors of efficiency of the computation and the programming implementation effort. C++ and Fortran implementation paradigms are compared and discussed from this point of view.

1 Introduction

Development of scientific applications for high performance distributed memory computer architectures is challenging. This challenge is particularly apparent during the programming process, when the developer often writes the application using constructs that are specific to the programming language and computer architecture used, but remotely related to the problem. The increasing complexity required in modeling scientific applications can only be controlled by using paradigms which ease the programming burden. We demonstrate how object oriented methods can address this issue on the example of a plasma Particle In Cell simulation.

We begin by comparing the organization of the Fortran and C++ programs that describe simulation of a beam-plasma instability experiment. In particular, we examine the process of converting the procedural Fortran codes into object oriented C++ versions. A comparative analysis of the simulation results obtained from the codes as well as their performance analysis are presented. Finally, the conclusions regarding choice of developing parallel programs in Fortran or C++ are offered.

2 The Plasma Particle In Cell Simulation Model

When a material is subjected to conditions under which the electrons are stripped off of the atoms, acquiring free motion, the mixture of heavy positively charged ions and fast electrons forms an ionized gas called a plasma. Ionization can be introduced by extreme

*This work is supported by NASA under Grant # NASA NGT-70334. The content does not necessarily reflect the position or policy of the U.S. Government. No official endorsements should be inferred or implied. Access to the Intel Paragon and Cray T3D at the Jet Propulsion Laboratory was provided by NASA's Offices of Aeronautics, Mission to Planet Earth and Space Science.

†Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180-3590, USA.

‡Department of Physics, University of California at Los Angeles, Los Angeles, CA 90024-1547, USA and NASA Jet Propulsion Laboratory, Pasadena CA 91109, USA.

heat, pressure or electric discharges. The free electrons can transport a current; thus fusion energy is an important area of plasma physics research, but more familiar examples include the aurora borealis, neon signs, the ionosphere, and solar winds.

The fundamentals of the Particle In Cell plasma simulation model are described in [1]. The model integrates in time the trajectories of large numbers of charged particles in their self-consistent electrostatic (coulomb) fields. The PIC method assumes that particles do not interact with each other directly, but through the fields which they produce according to the Maxwell's equations. For a one-dimensional simulation, all spatial variation is in the $\boldsymbol{x}$ direction, whereas in two-dimensions, motion occurs in the $\boldsymbol{x}$ and $\boldsymbol{y}$ directions. Particles can be located anywhere in the spatial domain; however, the field quantities are calculated on a fixed grid.

The computational cycle begins with the known initial particle positions $\boldsymbol{x}_i$ from which the charge density $\rho(\boldsymbol{x}_n)$ is found at the grid points $\boldsymbol{x}_n$ by interpolation,

$$\rho(\boldsymbol{x}_n) = \sum_i q_i S(\boldsymbol{x}_n - \boldsymbol{x}_i). \tag{1}$$

S is the particle shape function which replaces point charges with a finite size charge cloud to reduce close range collision effects. The particle charge is q_i with center $\boldsymbol{x}_i$ [5]. Next, the electric field $\boldsymbol{E}(\boldsymbol{x}_n)$ is found at the grid points by solving Poisson's Equation

$$\nabla \cdot \boldsymbol{E}(\boldsymbol{x}_n) = -\nabla^2 \phi = 4\pi \rho(\boldsymbol{x}_n), \tag{2}$$

using the Fast Fourier Transform. This electric field is then used to calculate the force on each particle whose trajectories are updated by integrating Newton's Law,

$$\begin{aligned} \frac{d\boldsymbol{v}_i}{dt} &= \frac{q_i}{m_i}\boldsymbol{E}(\boldsymbol{x}_i) \\ \frac{d\boldsymbol{x}_i}{dt} &= \boldsymbol{v}_i. \end{aligned} \tag{3}$$

(Note that we will not account for the magnetic field $\boldsymbol{B}$ in this simulation). The velocity and position of each particle is advanced in time using a time centered leap-frog scheme,

$$\begin{aligned} \boldsymbol{v}_i\left(t + \frac{\Delta t}{2}\right) &= \boldsymbol{v}_i\left(t - \frac{\Delta t}{2}\right) + \frac{F_i(t)}{m_i}\Delta t \\ \boldsymbol{x}_i(t + \Delta t) &= \boldsymbol{x}_i(t) + \boldsymbol{v}_i\left(t + \frac{\Delta t}{2}\right)\Delta t. \end{aligned} \tag{4}$$

This cycle then repeats for the duration of the simulation. Diagnostics are computed along the way and all lengths are normalized to the grid spacing. These lengths are related back to physical lengths later.

The General Concurrent Particle in Cell Algorithm described in [3] was used for the beam-plasma instability simulation. The GCPIC method partitions the particles and grid points among the N_p processors of the MIMD distributed memory machine. Each processor is assigned a subdomain and is responsible for the particles in its domain. During each time step a primary decomposition, which makes advancing particles in space efficient, and a secondary decomposition, which makes solving the field equations on the grid efficient, are used. When a particle moves from the current to the new partition, it is passed to the processor responsible for the new partition. The grid spacing is uniform and the particles are initially distributed uniformly. Initial motion of particles is described by Maxwellian

velocity with drift. Although the number of particles per processor will vary during this simulation, the load usually remains well balanced.

The beam-plasma instability experiment models a beam of electrons in a plasma which drives plasma waves to instability. Initially, there are two groups of electrons, a large population at rest (the background) and a smaller one moving at some nonzero velocity (the beam). An experiment such as this can be used to verify plasma theories and to study the time evolution of macroscopic quantities like potential and velocity distributions. Additionally, phase space diagrams of the particle positions over time can be examined to study the closed orbits of particle trapping in a potential well. More sophisticated experiments have been developed based on this model. However, for the purpose of examining Fortran verses C++ development, this experiment is sufficiently complex.

3 PIC Simulation in Fortran & C++

Most of the computational time in PIC simulation involves performing operations on particles and fields. Advancing the particles in space requires knowledge of the force due to the field. Additionally, computing the field requires knowledge of the charge contribution on the grid from the presence of particles. In Fortran, arrays maintain particle position and velocity information as well as grid data associated with charge density and directional forces. These arrays are passed by reference among functions which compute various energy diagnostics at each simulated time step. Although passing data arrays leads to a very efficient simulation in Fortran, the data interrelations are lost.

The simulation algorithm and its representation in Fortran displays certain characteristics of operations performed. Electrons, or particles, are advanced to new positions in space. Charge is deposited on the grid. The electric field is calculated on the grid and particles are distributed based on their type. In non-object oriented languages, we can write subroutines that perform these operations based on the availability of data; that is, less emphasis is placed on what the data represents and more on its availability.

In object oriented languages, however, a classification of the data and associated operations emphasizes its meaning. Thus, we need to look into what the data represents and find an appropriate set of classes and operations that support the user's idea of what the data means. For example, we should develop classes that utilize a grid with appropriate operations such as charge deposition. Classes that operate on particles with common and/or unique properties should also be introduced. Furthermore, objects of these classes should be integrated so that they can function together to perform specific operations. Various approaches have been presented regarding object oriented class design in plasma simulation [2, 4]. We discuss our design decisions in the next section.

3.1 The Object Oriented Class Design

When deciding how to represent a PIC program using an object oriented hierarchy, issues to be considered include:

- The impact of Fortran program structure on class design.
- The interdependence of efficiency and class design.
- The numerical reliability of C++ compared to Fortran.
- The utilization of various features of C++.

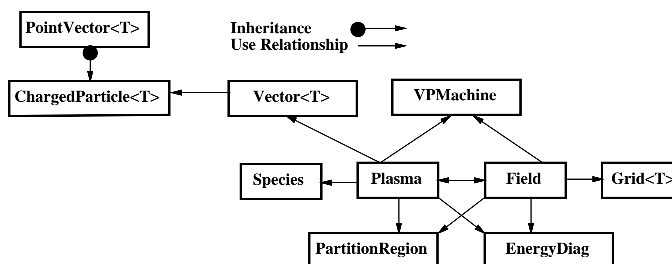

FIG. 1. Class Hierarchy

There will certainly be tradeoffs in the process of identifying and organizing classes from a Fortran based code. We overview this process while discussing the class organization of our program.

The C++ program uses templates to operate on a vector space of particles. A particle is described by a typed vector (specifically as either a `Vector1D<T>` or a `Vector2D<T>`) together with overloaded mathematical component point vector operations. The vector represents the position/velocity component in the corresponding dimensions. These vectors are inherited into a `ChargedParticle` class which enhances the physical description of a particle. Next, the vector space of charged particles is formed from the `Vector<ChargedParticle>` template class, allowing for the description of the plasma as a vector space of charged particles with vector operations on the collective group.

Actions such as distributing particles in space are performed on the charged particle vector space object, which is a collection of charged particles. Although each charged particle is a template object, care is taken to ensure that this organization is no less efficient than representing particles as arrays. Moreover, template representation allows the particles to be parameterized objects which is useful in specifying particle distribution properties and in computations associated with updating particle positions.

The `Species` class maintains specific information based on the group properties of particles, such as their thermal and drift velocities which distinguish them as background or beam particles. The `EnergyDiagnostic` class is used to collect and monitor plasma parameters associated with system energy. The `VirtualMachine` class parameterizes the parallel machine and therefore aids in portability since only the internal message passing calls need to be modified in this class. However, most of the computational effort is focused on the interactions between the electrostatic field and particles which result in the definition of the most complex classes, the `Field` and `Plasma` classes.

3.2 The Particle/Field Class Interaction

The field consists of a uniform computational grid which is distributed across processors. Since the Field is computed in Fourier Space there are actually two grids that compose the field, a scalar and a complex grid. Our Field class contains scalar and complex `Grid<T>` template classes as members. Since the field is partitioned across processors this class also contains a `PartitionRegion` object which maintains partitioning related information. Operations associated with depositing charge based on particle positions and calculating a uniform background ion density are members of the field class since they perform operations that modify the field.

Particles are treated as a collection in the vector space, therefore, a `Plasma` class was introduced to allow for collective operations. These include advancing particles and

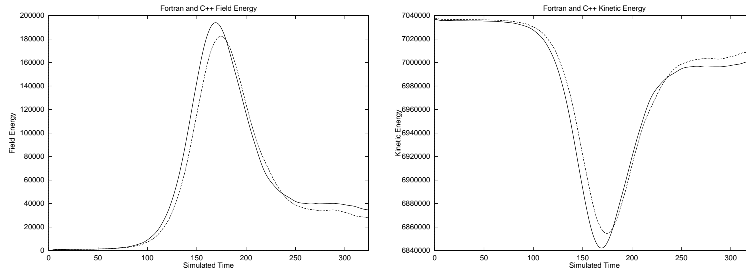

FIG. 2. Energy/Speed Distributions

updating their distribution across processors as they cross domain boundaries. Particles are also distributed, potentially in a different manner than the field, across processors, thus this class also maintains a `PartitionRegion` object. Additionally, particle space/velocity distributions can be specified.

It is useful to consider alternative organizations. In this code, we decided to bind operations that change the data associated with some concept to that class. Thus, modifying the grid charge density is a field operation. Similarly, moving particles among processors is a plasma operation. However, certain actions represent the union of particle/field interactions, such as advancing particles to new positions. We chose to view this as an operation on the particles under the influence of the field, therefore advancing particles is a plasma class operation. An alternative view would be to create a distinct class that operates on the collective interaction of particles and fields. This strategy would allow for particle and field classes to act specifically on particles and fields with a separate unified class used to operate on their interaction. Both approaches have merits and we are currently considering this alternative modeling approach. As sketch of the major portions of our class hierarchy is shown in Figure 1.

4 Simulation Results

In our beam-plasma instability experiment, we measure the field, kinetic and total energies of the system at each simulated time step. As mentioned, we begin with two populations of electrons; one large group at rest and a smaller group in motion. Our 1D Paragon simulations consisted of 4,096,000 background electrons and 409,600 beam electrons with 16384 grid points. The 2D SP1 simulations consisted of 3,276,800 background and 294,912 beam particles with 32768 grid points. Since the original Fortran codes have been well benchmarked [3], we will restrict our performance overview to these rather arbitrarily selected cases.

Examining the energy diagnostics for the SP1 two-dimensional simulations with 3,571712 particles on 16 processors, Figure 2 shows that the curves correspond for the Fortran and C++ versions, illustrating the field/kinetic energy exchange. Additionally, we illustrate the plasma phase space. The Fortran version completed in 802 seconds while the C++ version finshed in 1,228 seconds. The Paragon one-dimensional 4,505,600 simulation on 32 processors gives completion times of 231 seconds and 377 seconds for Fortran and C++ respectively, which also is reasonably competitive.

4.1 Implementation Issues

C++ compilers are still very much in the evolutionary stage. We used the GNU g++ compiler v2.4.5 for the one-dimensional simulation results we have presented on the Intel Paragon. When the g++ compiler was upgraded to v2.5.7 the *identical* programs compiled correctly, yet the energy diagnostics reported were completely incorrect. The two-dimensional code was easily ported to the Paragon from the SP1, however we could observe that template references under g++ v2.5.7 introduced numerical errors into `ChargedParticle` vector operations when advancing particles. The identical references using xlC performed correctly. The Cray T3D C++ compiler, which is in beta release at this writing, could not instantiate templates correctly. Interestingly enough, however, the identical program did build correctly on the Cray YMP. Cray support is examining this issue at the time of this writing. The IBM xlC C++ compiler was used during development on the SP1. Both the SP1 and xlC compiler performed extermely well.

This code was easily ported among machines and compilers. However, lack of standardization regarding C++ template instantiation has been an issue of concern. Templates provide for many organizational advantages, but outstanding implementation issues currently limit their usefulness.

5 Conclusion

We have given an overview of the design of C++ skeleton particle simulation codes based on existing Fortran codes. The design concepts involved in reorganizing the Fortran program into C++ have been discussed. Additionally, we have given performance results which indicate that the execution speed of C++ may be acceptable, given the organizational advantages. The codes were designed with both execution and implementation scalability in mind. The template classes aided in this extension since moving from one to two-dimensions for particle vector spaces required just modification of the parameterized type.

Programming in an object oriented manner takes practice and patience. As numerical classes are introduced into the language and as new techniques are found to improve the efficiency of C++ programs, the benefits of object oriented design will influence scalable high performance computing. Our ongoing research into using C++ for runtime efficiency of unstructured and irregular parallel computations, such as in more complex plasma simulations, is the focus of our future efforts. The ability to reuse existing software and to develop computation kernels represents a growing need as high performance computing becomes more complex. Object oriented methods can help to achieve this goal.

References

[1] C. K. Birdsall and A. B. Langdon, *Plasma Physics via Computer Simulation*, The Adam Hilger Series on Plasma Physics, Adam Hilger, New York, 1991.

[2] S. W. Haney and J. A. Crotinger, *C++ Proves Useful in Writing a Tokamak Systems Code*, J. Computers in Physics, 6 (1991), pp. 450–455.

[3] P. C. Liewer and V. K. Decyk, *A General Concurrent Algorithm for Plasma Particle-in-Cell Simulation Codes*, J. of Computational Physics, 85 (1989), pp. 302–322.

[4] J. V. W. Reynders, *Object-Oriented Particle Simulation on Parallel Computers*, in 15th International Conference on the Numerical Simulation of Plasmas, King of Prussia, Pennsylvania, 1994, Princetion University Plasma Physics Laboratory and U.S. Department of Energy Office of Fusion Research, pp. 1B2 1–4.

[5] T. Tajima, *Computational Plasma Physics: With Applications to Fusion and Astrophysics*, Frontiers in Physics Lecture Note Series, Addison Wesley, Redwood City, CA, 1989.