Proc. Computing in High Energy Physics'95, Rio de Janeiro, Brazil, September, 1995,
World Scientific, Singapore, 1996, pp. 26-30

Experiences with Object Oriented Parallel Plasma PIC Simulations

Viktor K. Decyk

*Physics Department, University of California, Los Angeles (UCLA),
Los Angeles California, 90024-1547, USA
and Jet Propulsion Laboratory/California Institute of Technology,
Pasadena, California 91109, USA*

Charles D. Norton and Boleslaw K. Szymanski

*Department of Computer Science, Amos Eaton Hall
Rensselaer Polytechnic Institute, Troy, New York, 12180-3590, USA*

1 Introduction

The Numerical Tokamak Project is a High Performance Computing project involving nine institutions, sponsored by the U. S. Department of Energy[1]. Its goal is to model a fusion energy device known as a tokamak in order to understand and predict the transport of particles and energy in these devices. Tokamaks, which are toroidal in shape, confine the plasma with a combination of an external toroidal magnetic field and a self-generated poloidal magnetic field. The plasma confinement in these devices is not well understood and is worse than desired.

One of the two computer models used in this project is a gyrokinetic code, which is a reduced particle-in-cell (PIC) code that follows the trajectories of guiding centers of particles, neglecting the rapid rotation around the magnetic field. Particle-in-Cell codes integrate the trajectories of many particles subject to electromagnetic forces, both external and self-generated. These forces are calculated from a set of field equations (usually Maxwell's equations or a subset) on a grid. The particle's coordinates are described by continuous variables. The source terms in Maxwell's equations (charge and/or current density) are calculated on a grid by inverse interpolation. After the field equations have been solved on the grid, the forces on the particles are found by interpolation from the grid.

The size of the computation required is very large. For the tokamaks of interest, the number of cells required is about $500 \times 500 \times 128$ or $1000 \times 1000 \times 128$. Since one needs at least 10 particles per cell to obtain reasonable statistics, the code must be capable of following at least several hundred million particles over thousands of time steps. The only computers currently capable of handling such large problems are the Massively Parallel Processors (MPPs), thus parallel computing is an important aspect of the Numerical Tokamak Project. There are other aspects which make the project complex. These include management of large amounts of data and its visualization, collaboration among widely scattered scientists, and an increasing desire to add more realism.

This project is too ambitious to be completed by one individual and requires a larger team. As a result, we have increased our collaboration with computer

Table 1: Fortran 77 and Fortran 90 Comparative Examples

Fortran 77 Code Segment	**Fortran 90 Code Segment**
dimension part(idim,np) dimension fx(nx) data qm,qbm,dt /-1.,-1.,.2/ call push1(part,qbm,eke,idim,np,fx,nx,dt) call dpost1(part,qm,idim,np,q,nx)	use plasma_module type (species) :: electrons, ions type (fields) :: charge_density, efield real :: dt = .2 call plasma_push1(electrons,efield,dt) call plasma_dpost1(electrons,charge_density)
F90 Derived Type for Complex	**F90 Declaration of Complex Variable**
type complex real :: x, y end type complex	type (complex) c

scientists and others who have the expertise that is required for the success of this project. This talk will focus on one such collaboration with computer scientists at Rensselaer Polytechnic Institute in Troy, New York.

2 Object Oriented Concepts for Fortran Users

Since the Numerical Tokamak project is large and ambitious, several members of this project and others are exploring the usefulness of object-oriented techniques in managing large, complex PIC codes[2,3,4]. Our use of these techniques is currently exploratory, designed to learn how to use them effectively and to understand what their main value (if any) is. As part of this exploration, we converted several PIC codes from their original Fortran 77 to C++ and to Fortran 90.

C++ as an object-oriented language has been attracting attention in the physics community for some time now[5]. However, it is not commonly known that Fortran90 also supports many object-oriented features, in addition to its more well-known array processing features[6]. Since many physicists are familiar with Fortran, we thought it might be useful to illustrate some of the most important concepts of object-oriented programming by referring to Fortran. Unfortunately, the vocabulary used by Fortran 90 and C++ for the same concepts differs. Thus for each concept, we provide the terminology used by both languages.

Table 1 shows a few typical lines from a Fortran 77 particle code. The `push1` subroutine advances the particles, whose coordinates are stored in the array called `part`, by interpolating from the electric field array called `fx`, using the time step `dt`. The arguments in the subroutines pass information about the particles and fields, such as the array names, their dimensions, and constants such as the charge and charge/mass ratio for the particles. It would make the code clearer if all the information about particles were stored together with one name (encapsulated), and all the information about fields in another, and we could always refer to them together, as shown in the Fortran 90 code in Table 1.

Fortran 90 allows for encapsulation of data using derived types (which are called structures in C). Fortran 77 natively supports a number of data types such as integer, real, complex, and character. In Fortran 90, one can add arbitrary data types. Table 1 shows a complex type and the declaration of a complex variable `c`. In this manner, we can define a species type to represent particles. This species type could contain the particle array, and all the constants which describe particles, such as their charge and their number into a single structure. We can do the same for fields.

This is all nice, but how do we tell the Fortran program about this new data type? The best way is to put the definition of this new type into what Fortran 90 calls a module. This module can then be "used" in other procedures. This is like an include file in Fortran 77, except the "module file" is not external. Furthermore, Fortran 90 does not natively know how to operate on the new data type. The user must define these operations in procedures (such as the new plasma_push1). Thus it makes sense to put these new procedures in the same module which defines the new data type. This idea of combining new data structures with the procedures which can operate on them is central to the idea of object oriented programming, and is called a class in C++. Data types and procedures within a module are accessible only to procedures which use the module. For added safety, some items in a module can be further restricted as `private` where the data types are accessible only from within defining module's procedures. This is called encapsulation in C++.

Note that the modules contain the data type definition, but they do not necessarily contain the data (the variable) itself. In the example above, we have declared two variables of type `fields` and `species` in the main program. The actual variable is called an object (or an instantiation of the class) in C++.

It is useful to allow a hierarchy of modules. For example, for the fields module, which knows about the fields data type, one can include all the procedures which operate only on fields, but not on particles, e.g. setting the initial charge density data to zero. Similarly, the species module can include procedures which operate only on data of type species, such as assigning initial particle coordinates. However, the procedure plasma_push1 needs to know about both species and fields, so we can construct another module, called plasma, which contains procedures that operate on both. This new plasma module can obtain information about the species and fields data types and procedures by `using` the species and fields modules. This is called inheritance in C++. One can also use (or inherit) selectively a portion of the module.

In Fortran, an operation is evaluated differently depending on the type of the operands. For example, a/b will give a different result if a and b are integers than if they are real or complex. Fortran 90 extends this concept to derived types. Thus it is possible to have a `generic` push subroutine, which will operate differently on ions than on electrons. This is done by declaring ions and electrons to be of different derived types and defining what procedure the generic push will execute for each type. Such generic functions are called virtual functions in C++.

Converting a code from Fortran 77 to Fortran 90 using these concepts resulted in a main program that is very simple and elegant. Subroutines (many of which we placed in modules) underwent minor changes. This new organization simplifies the addition of new features, such as those required for parallel processing.

Table 2: IBM RS6000 Sequential and IBM SP2 Parallel Benchmarks

1D Sequential Benchmark – 450,000 Particles					
Fortran 77	245.49s	Fortran 90	364.25s	C++	508.00s
3D Parallel Benchmark – 7,962,624 Particles					
Fortran 77	1649.00s	Fortran 90	N/A	C++	2797.00s

3 Program Design with Object-Oriented Techniques

The object-oriented paradigm encourages an application based view of programming to emphasize clarity and reuse of software components. When properly designed, classes for sequential computations can be extended directly for use on MPP architectures (with the support of additional classes). For example, the sequential codes have no support for message passing or managing distributed data. We added a virtual parallel machine class to provide object-based communication consistently across various MPP architectures. The sequential field class was reused and extended with new operations to support distributed data. This included routines to replicate and transport border field data to neighbor processors to reduce off-processor references. Additionally, classes which contain distributed data now have access to partition objects which maintain border information. Integrating such features into the parallel codes was straightforward since the components of the program were clearly defined by encapsulation.

C++ provides features beyond object-oriented techniques to support programming, such as templates. Template classes use objects as parameters in their definition. We used templates to represent the plasma particles and the distributed field to simplify moving from two to three-dimensional C++ codes. By redefining our particle class as a vector template class, we can specify particles as vectors in any dimension. Vector operations on particles can then remained unchanged when moving to higher dimensional codes; only the template parameter specifying the dimension of the particle is required at compile-time. The definition of multidimensional data structures can be handled in the same manner. This kind of modeling would be difficult to emulate in Fortran.

4 Performance Issues

Table 2 shows simple benchmark cases of a one-dimensional sequential and three-dimensional parallel plasma code. The Fortran 90 sequential code accesses particle data through module functions corresponding to the C++ code use of class member functions. Since these calls were not inlined in our Fortran 90 program, this contributed significantly to the performance overhead observed. Reorganization of the code to make this data directly accessible reduced the Fortran 90 execution time slightly below that of the Fortran 77 program. Regarding the parallel three-dimensional Fortran 77 and C++ codes, we believe that Fortran 90 and HPF may improve performance over C++ and improve reuse and clarity over Fortran 77.

5 Conclusions

There are a number of benefits of the object-oriented approach. One obvious benefit is that the code looks much simpler and easier to read. Part of this clarity arises because details of procedures can be hidden. A further advantage is that it becomes easy to add new data types and operations without causing unintended side-effects, since data and associated operations on the data are encapsulated in classes (or modules). Thus extending or modifying the code becomes much easier. This encourages experimenting with new ideas and results in better science. Another advantage is enhanced collaboration. If different scientists or groups can agree on common data structures and associated procedures, then code development can proceed independently without fear of incompatibility.

One of the disadvantages of the object-oriented approach, is the effort needed to learn new ways of doing things. Another is degraded performance, but for large projects the advantages of increased clarity and modifiability of the programs often outweigh these disadvantages.

Of the two object-oriented approaches investigated, C++ has some advantages compared to Fortran 90. There is a large community of practitioners to help in learning the new approach. It is also more modern and introduces more new ideas.

However, Fortran 90 also has a number of advantages. It is backward compatible with Fortran 77, so the transition is easier, and one can proceed incrementally, without disruption or great investments in recoding. It supports high level array constructs, which are useful in physics. It is closely related to High Performance Fortran and thus is a natural migration path to parallel computers. Fortran 90 generally produces code which executes faster. Finally, the environment is more stable (because it is standardized and not rapidly evolving).

Acknowledgments

The first author is supported by NASA HPCC, NSF and DOE. The remaining authors are supported by the NASA Graduate Student Researchers Program.

References

1. B. I. Cohen, et. al. *Computer Physics Communications*, 87(1&2):1–15, May II 1995.
2. C. D. Norton, B. K. Szymanski, and V. K. Decyk. *To appear in Communications of the ACM*, 38(10), October 1995.
3. J. V. W. Reynders, D. W. Forslund, P. J. Hinker, M. Tholburn, D. G. Kilman, and W. F. Humphrey. *Computer Physics Communications*, 87(1&2):212–224, May II 1995.
4. J. P. Verboncoeur, A. B. Langdon, and N. T. Gladd. *Computer Physics Communications*, 87(1&2):199–211, May II 1995.
5. B. Stroustrup. *The C++ Programming Language.* Addison–Wesley, Reading, MA, second edition, 1991.
6. T. M. R. Ellis, I. R. Philips, and T. M. Lahey. *Fortran 90 Programming.* Addison–Wesley, Reading, M.A., 1994.