High Performance Object-Oriented Scientific Programming in Fortran 90

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

Abstract

We illustrate how **Fortran 90 supports object-oriented concepts by example of plasma particle computations on the IBM SP. Our experience shows that Fortran 90 and object-oriented methodology give high performance while providing a bridge from Fortran 77 legacy codes to modern programming principles. All of our object-oriented Fortran 90 codes execute more quickly than the equivalent C++ versions, yet the abstraction modeling capabilities used for scientific programming are comparably powerful,**

1 Introduction

Computer simulations are very useful for understanding and predicting the transport of particles and energy in fusion energy devices called tokamaks [1]. 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 the Numerical Turbulent Transport Project is a gyrokinetic code—a reduced particle-in-cell (PIC) code that follows the trajectories of guiding centers of particles, neglecting the rapid rotation around the magnetic field. PIC codes integrate the trajectories of many particles subject to electro-magnetic 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 interpolation. After the field equations have been solved on the grid, the forces on the particles are found by interpolation from the grid.

The General Concurrent PIC Algorithm [8], which partitions particles and fields among the processors of a distributed-memory supercomputer, can be programmed using an SPMD design approach. Although the Fortran 77 versions of these programs have been well-benchmarked [4], there is an increasing interest in the scientific community to apply object-oriented principles in scientific programming [12]. In a three-dimensional PIC program,

*Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109-8O99. The author's research was conducted while holding a National Research Council Research Associateship. Email: nortonc@olympic.jpl.nasa.gov

†Department of Physics and Astronomy, " University of California at Los Angeles, Los Angeles, CA 90024-1547 and Jet Propulsion Laboratory, California Institute of Technology, Pasadena CA 91 10SI-8099. The author's research was sponsored by USDOE, NSF, and under a contract with NASA. Email: decyk@physics.ucla.edu

‡ Department of Computer Science and Scientific Computation Research Center (SCOREC), Rensselaer Polytechnic Institute, Troy, NY 12180-3590. The author's research was sponsored under grants CCR-9216053 and CCR-9527151 Email: szymansk@cs.rpi.edu

advancing particles in space and depositing charge to the field might look like the following in Fortran 77.

```
dimension part (idimp ,npmax) , q(nx ,ny,nzpmx)
dimension fx(nx, ny, nzpmx) , fy(nx, ny, nzpmx), fz(nx, ny,nzpmx)
data qme,dt /-1.,.2/
call push3 (part,fx,fy,fz,npp,noff ,qtme,dt,wke,nx,ny,idimp,npmax,nzpmx)
call dpost3 (part,q,npp,noff,qme,nx,ny,idimp,npmax,nzpmx)
```

The particles (part) are described by continuous variables. The charge density field (q) and the electric force field (fx,fy,fz) are represented by discrete grids. All ofthese components are slab-partitioned across the processors. The most time consuming operations ofpushing particles (push3) and depositing charge (dpost3) contain parameters involving particles and fields with additional components required by the computation. Ideally, it would be preferable to introduce abstractions that clarify this representation.

```
USE partition.module
USE plasma_module
TYPE (species3d) :: electrons
TYPE (sfields3d):: charge. density
TYPE (vfields3d)::efield
TYPE (slabs) :: edges
real  :: dt = .2
call plasma_ push3 (electrons, efield, edges, dt)
call plasma_dpost3 (electrons, charge_density, edges)
```

In this Fortran 90 version the electron species properties have been encapsulated into a species ***derived-type.*** Similarly, properties of the scalar charge density and vector electric field have each been encapsulated. By use-association of a *module*, access to routines and datathat defineoperations on the particles and fields are now available tothemainprogram. Additionally, the creation of abstractions—such as the fields----may have come from existing abstractions, but this is all information hidden behind various interfaces. The abstraction modeling capabilities of this modern language are attractive. In this paper we consider the benefits of applying object-oriented principles to the new constructs of Fortran 90.

2 Abstraction Modeling in Fortran 90

The new features of Fortran 90, many of which have not been presented in the simple example above, encourage the creation of abstract data types, encapsulation, information hiding, inheritance, function overloading, and many features beyond array-syntax to ensure the safe development of advanced programs. Many of these ideas support object-oriented concepts [5]. Furthermore, since Fortran 90 is a subset of High Performance Fortran, a migration path to parallel computation using these techniques exists as well.

The Fortran 90 programming language contains many new features while remaining backward compatible to Fortran 77 [6]. The most important features for our purposes include ***derived-types, pointers, dynamic memory operations, army-syntax and related intrinsic functions, modules, overloading,*** **generic** ***procedures, and use-association.*** All of these concepts were applied in writing a two-dimensional object-oriented PIC program in Fort ran 90 [10], therefore, we will describe how that program was extended into three-dimensions. A much more complete description of how Fortran 90 statements map into

```
MODULE species_class
  USE distribution-class          ! bring in other class features
  USE slab_partition_class
  IMPLICIT NONE
  TYPE particle3d                 ! derived-type
    PRIVATE
    real :: x, y, z, vx, Vy, Vz
  END TYPE particle3d
  TYPE species3d                  ! derived-type
    real ::qm, qbm, ek
    integer ::nop, npp
    TYPE (particle3d), DIMENSION (:), POINTER ::p
  END TYPE species3d
CONTAINS
  SUBROUTINE spec_dist(this,edges,distf,noff)        ! member function
    TYPE (species3d), INTENT (out) :: this
    TYPE (slab), INTENT(in):: edges
    TYPE (distfcn), INTENT (in):: distf
    INTEGER, INTENT (in):: noff
    ! subroutine body...
  END SUBROUTINE spec_dist
  ! additional member functions. . .
END MODULE species-class
```

FIG. 1. *A Fortran 90 module, for the species class, supporting various object-oriented concepts. Features of other classes are made available by use-association, objects are defined by Fortran 90 derived-type variables, and member functions operating on the objects are specified within the contains statement.*

object-oriented concepts are described in related publications [5, 9, 10]. Nevertheless, figure 1 shows asection ofthe species class that illustrates some ofthese features.

A Fortran 90 module may contain data, type declarations, and related routines. Fortran 90 derived-types, similar to structures in the C programming language, allow new types to be created from intrinsic and previously defined derived-types.[1] Module components are public by default, but some (or all) of the components can be private. Additionally, derived-types can have private components, seen in the particle3d type, restricting access to module member functions.

The use statement makes components of existing modules available within the scope they are used. This allows direct support for some object-oriented features, such as inheritance by composition, as well as the emulation ofother features, like inheritanceby sub-typing. Some languages (including C++) define inheritance so that sub-typing—the ability ofa derived class object to assume the type ofabase class object—is automatically applied. This form of inheritance is somewhat restrictive, compared to a more "open" definition such as composition inheritance—defining new class objects as incremental changes to existing classes where sub-typing is not necessarily preserved, Froma theoretical

[1] Fortran 90 introduces the term derived-type to mean abstract data type, which shouldn't be confused with "derived" used in other contexts (such as derived class) in an object-oriented class hierarchy.

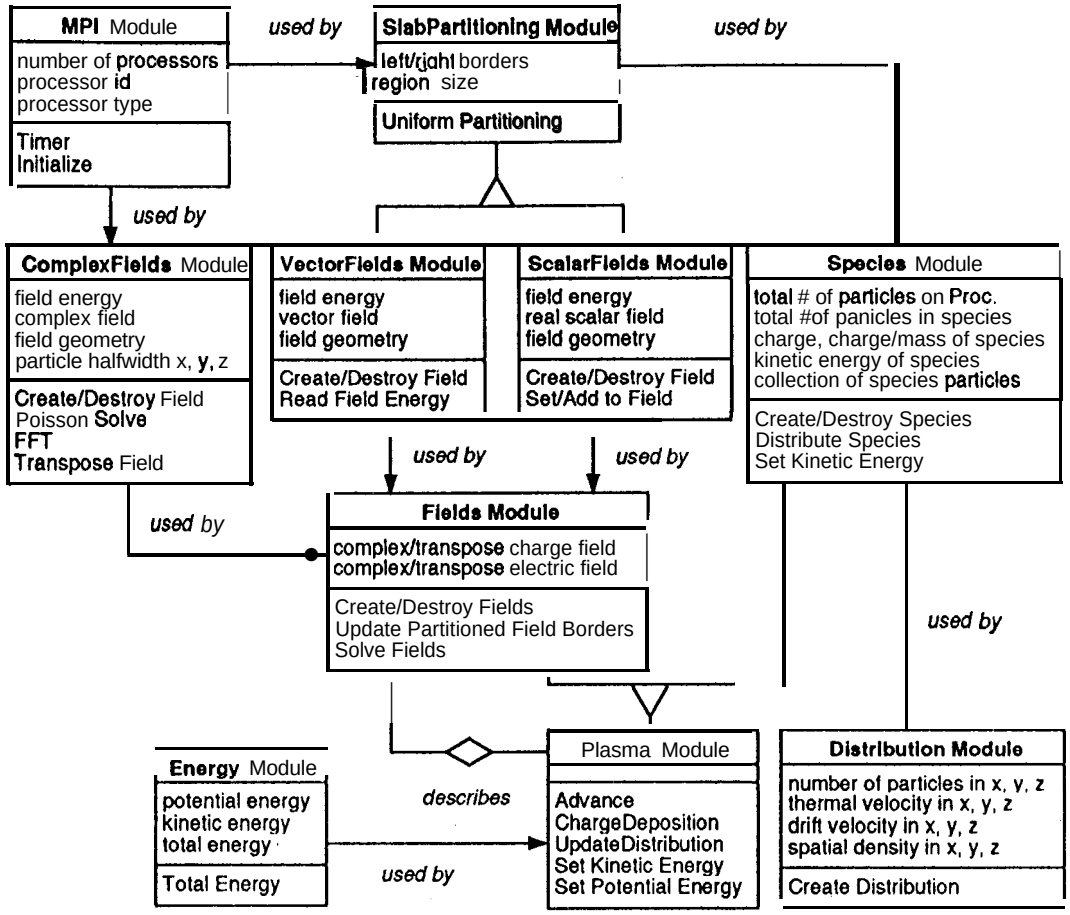

FIG. 2. *The model design for the object-oriented Fortran* 90 *parallel program.*

viewpoint some believe that combining sub-typing and inheritance leads to misconceptions, in fact, they should be separate issues [3, 7]. In general, we have found that composition inheritance is more useful than sub-typing inheritance in scientific programming [9].

Fortran 90 is a statically-typed language, all references to types must be resolved at compile-t ime, indeed, all t ypcs must be unambiguously (or uniquely) defined. Formal parameters to routines defined within a module automatically contain implicit intcrfacc specifications allowing t hc compiler to type-check routine arguments at compile-t imc. (An explicit interface block can be introduced for routines not defined within modules.) Syntactically, objects must be present in the routine call, so we make the first argument the object performing the operation and call it "this" by convention. Formal parameters may have the intent attribute determining the read-only, write-only, or read-write status of arguments to the routine. A variety of array types supporting complete and array subsection operations are included in the language standard.

3 Object-Oriented PIC Programs in Fortran 90 and C++

The original Fortran 77 programs were reorganized into an object-oriented form and implemented in Fortran 90 and C+-+. Figure 2 shows t hc Fortran 90 organization using t hc OMT notation [11]. The structure of the three-dimensional program is identical to the two-dimensional version, but the internal encapsulated features have been modified for three-

dimensional programs. The ScalarFields and VectorFields modules inherit partitioning information from t hc SlabPart it ioning module. Interprocessor communicant ion feat ures are provided by the MPI Module. This module binds the Fortran 77 MPI library calls to Fortran 90 through interface declarations. The particle species---defined by the Species **Module wit h particle distribut ions specified by the Distribution Module--is also partitioned, however, information known by the slab partitioning module is simply used by the species, so an inheritance relation is not defined. The Fields Module contains routines that operate on the scalar charge density field, vector electric field, and complex fields (used** by the FFT's and Poisson's Equation solver). The Plasma Module contains no data, only routines that operate on particles and fields including the **push3** (Advance) and dpost3 (Charge Deposition) operations described earlier.

The derived-types, which are the **class** data members in Fortran 90 programming, are encapsulated in the module definitions. Variables of these types form the '{objects" used in Fortran 90 programs, Modules may contain many derived-types and these may contribute to the definition of new derived-types, as we have seen in figure 1. Hence, the routines in the Fortran 90 contains statement may operate on any number of the derived-types contained therein. This approach is different than in a language such as C++ where a single object represents an entire class. These differences have implications on how scientific programs are modeled using object-oriented techniques [9].

Figure 3 shows the major operations of the main program for the three-dimensional code. Since the partition and plasma modules use other modules in their definition all of the objects can be created from use-association of these modules. In C++, object constructors combine declaration and initialization into a single statement, whereas in Fortran 90 they must be done separately. Fort ran 90, however, does provide an array constructor syntax. All of the major operations involve high-level concepts, while the implementation details are encapsulated in module/class definitions.

One aspect some consider important in object-oriented programming is the use of run-time polymorphism, or dynamic-dispatching. The technique, which introduces a run-time penalty, allows an operation **to bc applied to objects sharing a sub-typing inheritance relat ionship, however, the action performed varies with the object's type. Dynamic-dispatching was not needed in** the C++ or Fortran 90 programs (all objects could be resolved at compile-time). Nevert heless, **we have found that object-oriented techniques are beneficial even when some aspects are not applied, or find limited usefulness.**

4 Performance

Table 1 shows the performance for two-dimensional and three-dimensional parallel programs on the IBM SP2. The most remarkable aspect is that the object-oriented Fortran 90 programs were competitive with the Fortran 77 versions, and always more efficient than the C++ programs.

The **three-dimensional Fortran 90 program outperformed the Fortran 77 version** due to the improved cache-utilization of clustering field components into a single Fortran 90 derived type rather than storing each field dimension in separate Fortran 77 arrays. For example, a 294,912 particle simulation on 4 processors completed in 367.88 seconds in Fortran 90 while the Fortran 77 version finished in 411.00 seconds. The Fortran 90 program used 559.64 milliseconds per call combined to advance particle positions and velocities in **push3 and deposit charge in dpost3 while the Fortran 77 program used a total of 685.66 milliseconds for the same operations. This represents 64.7% of the total cumulative time of**

```
PROGRAM beps3k.
  USE partition_class,plasma_class
  TYPE (distf3d) :: backdf, beamdf
  TYPE (species3d):: electrons
  TYPE (sfields3d) :: cdensity
  TYPE (vfields3d) :: efield
  TYPE (energy) :: energ
  TYPE (slab) :: edges
  call MPI_INIT(ierror)
  call species_init(electrons,qme,qbme,np,nvp)   ! initialize particles
  call fields_init(cdensity ,nx,ny,nz,nvp)       ! initialize fields with
  call fields_init(efield,nx,ny,nz,nvp)          ! generic procedures
  call plasma_dpost3(electrons,cdensity,edges) ! initial charge deposit
  call sfields_add(cdensity,edges,qi0)           ! ion background charge
  DO itime=1,500
       call fields_solve(cdensity,efield)            ! talc. force/charge
       call plasma_getpe(energ,efield)               ! efield energy
       call plasma_push3(electrons,efield,edges,dt) ! push particles
       call plasma_pmove3(electrons,edges)           ! move across pEs
       call plasma_dpost3(electrons,cdensity,edges)
       call sfields_add(cdensity,edges,qi0)
       call plasma_getke(energ,electrons)        ! particle energy
       call energy_gette(energ)                  ! energy diagnostic
  END DO
  call fields_destroy(efield)              ! destroy dynamic objects
  call fields_destroy(cdensity)
  call species_destroy(electrons)
  MPI_FINALIZE(ierror)
END PROGRAM beps3k
```

FIG. 3. *Sections of the Fortran 90* main *program for the three-dimensional object-oriented parallel* **simulation** *program. Objects are declared and initialized from derived- types defined within modules. Fortran 90 generic procedures allow fields of different types to be created using* **a common routine name.** *The objects are the first argument to module member functions, by convention.*

the simulation for those two functions in Fortran 90 compared to 71.0% in Fortran 77. On a problem where the fields fit into the entire cache, using 15,552 particles on 4 processors, the Fortran90 version completed in 22.77 seconds while the Fortran 77 version finished in 20.96 seconds. The Fortran 90 program used 24.05 milliseconds per call combined in the particle advance and charge deposition routines while the Fortran77 program used 23.84 milliseconds per call combined. In both cases, the xlf90 compiler was used.

Another aspect of performance is development time. The two-dimensional object-oriented Fortran 90 program was extended by incrementally modifying features of the two-dimensional Fortran 77 version. The benefit ofthis approach is that redevelopment in a new language, possibly C++, was not required. This simplified testing and allowed the three-dimensional object-oriented Fortran 90 version to be programmed over aweekend (by extending the two-dimensional object-oriented Fortran 90 version). Granted, these are

TABLE 1

Performance characteristics for a beam-plasma instability experiment. IBM SP2 (66.7 MHz clock per thin-node, 40 MB/see network user-space high-performance switch protocol, and AIX 4. 1) with 32 processors.

Machine	PEs	Language	Compiler	Particles	Time (sec.)
Two-Dimensional Program					
IBM SP2	32	Fortran 77	IBM xlf90	3,571,712	193.52
IBM SP2	32	Fortran 77	IBM xlf	3,571,712	195.08
IBM SP2	32	Fortran 90	IBM xlf90	3,571,712	202.88
IBM SP2	32	C++	IBM xlC	3,571,712	359.00
Three-Dimensional Program					
IBM SP2	32	Fortran 77	IBM xlf90	7,962,624	1548.71
IBM SP2	32	Fortran 77	IBM xlf	7,962,624	1550.14
IBM SP2	32	Fortran 90	IBM xlf90	7,962,624	1339.91
IBM SP2	32	C++	IBM xlC	7,962,624	2797,00

skeleton programs, but the encapsulated features limited where changes were introduced and the main program was essentially identical to the two-dimensional object-oriented Fortran 90 version. When considering the use of object-oriented methodology for existing programs developed in Fortran 77, this approach may be more beneficial than trying to learn a new language only to gain abstraction in programming, particularly when the performance may not be as competitive or the user experience as advanced.

5 Conclusion

We have entered a period of time where the ability to produce scientific software that is reliable, portable, efficient, safe, and well-understood, is as important as efforts to construct supercomputers with sufficient sustained peak-performance and memory requirements to produce new science. Although we have used Fortran 90 in the context of object-oriented programming, the fundamental features can be beneficial even if these concepts are not applied. Although Fortran 90 was not designed as an object-oriented language, we are not the first to develop an object-oriented methodology for a language not explicitly object-oriented [2, 11]. Our experience indicates that it is possible to use Fortran 90 features to model many aspects of explicitly object-oriented languages, such as C++ [5]. However, such modeling can be tedious at times since some features, such as subtyping inheritance, must be constructed explicitly in Fortran 90. Efforts to formally introduce object-oriented language features in Fortran may be considered in future language draft proposals. Nevertheless, the benefits of abstraction with performance comparable to the original Fortran 77 programs makes Fortran 90 very appealing for object-oriented programming.

Current efforts include characterizing the features of Fortran 90 that can affect performance when object-oriented approaches are used. Some preliminary work in this area indicates that compilers exhibit varying levels of performance when array operations, use-association, and pointers within derived-types are applied, to name a few. Additional information beyond the scope of this paper is available on the Internet (http: //www.cs.rpi.edu/~szymansk/oof90.html).

Acknowledgments We appreciate the support of Robert D. Ferraro, associate project manager of NASA HPCC and Tom Cwik, technical group supervisor, Jet Propulsion Laboratory, California Institute of Technology. Additionally, we thank the Scientific Computation Research Center at Rensselaer Polytechnic Institute for access to the IBM SP2.

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] G. Booth, Object-Oriented *Development,* IEEE Transactions on Software Engineering, SE-12 (1986), pp. 211-221.

[3] W. Cook, W. Hill, and P. Canning, *Inheritance Is Not Subtyping,* in Proc. 17th ACM Symposium on Principles of Programming Languages, January 1990, pp. 125-135.

[4] V. K. Decyk, *Skeleton PIC Codes for Parallel Computers,* Computer Physics Communications, 87 (1995), pp. 87-94.

[5] V. K. Decyk, C. D. Norton, and B. K. Szymanski, *Introduction to Object-Oriented Concepts Using Fortran 90.* Submitted to Computers in Physics, July 1996.

[6] T. M. R, Ellis, I. R. Philips, and T. M. Lahey, *Fortran* 90 *Programming,* Addison-Wesley, Reading, MA, 1994.

[7] K. Fisher and J. C. Mitchell, *Notes on typed* object-oriented *programming,* in Theoretical Aspects of Computer Software, M. Hagiya and J. C. Mitchell, eds., Springer LNCS 789, 1994. Proc. of International Symp. TACS '94, Sendai, Japan - April 1994.

[8] 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.

[9] C. D. Norton, *Object Oriented Programming Paradigms* in *Scientific Computing,* Phi) thesis, Rensslеaer Polytechnic Institute, Troy, New York, August 1996. UMI Company.

[10] C. D. Norton, V. K. Decyk, and B. K. Szymanski, *On Parallel Object Oriented Programming in Fortran 90,* ACM SIGAPP Applied Computing Review, 4 (1996), pp. 27-31.

[11] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design,* Prentice Hall, Englewood Cliffs, NJ, 1991.

[12] B. K. Szymanski and C. D. Norton, eds., Special *Issue on Parallel Object-Oriented Programming,* vol. 4, ACM SIGAPP Applied Computing Review, Spring 1996. http: //www.acm.org/sigapp/acr/index.html.