ACM Applied Computing Review, **4**(1), Spring, 1996, pp. 27-31

On Parallel Object Oriented Programming in Fortran 90

Charles D. Norton[1], Viktor K. Decyk[2] and Boleslaw K. Szymanski[1]
[1]Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York, 12180-3590, USA
[2]Physics Department
University of California, Los Angeles
Los Angeles, California 90095-1547, USA

Abstract

The C++ programming language [6, 10] is well-known for its support of object oriented concepts, useful in abstraction modeling. Containing many important features, its popularity is growing with a new generation of scientists anxious to bring clarity and flexibility to their programming efforts. Nevertheless, most of the scientific applications in development and use today are based on Fortran, the most popular language for scientific programming.

Fortran is not a static language, it has continually evolved to include the most recent proven ideas and concepts garnered from other programming languages. Until recently, many of the most modern features were not available, complicating abstraction modeling for large scale development projects. This can make software difficult to comprehend, unsafe and potentially useless. The emergence of Fortran 90 [3] has dramatically changed the prospects of Fortran programming. Not only are many of the most modern aspects of programming language techniques included in the standard, there are also specific new additions that will undoubtedly affect the next generation of all languages used in scientific programming [8].

1 Why Fortran 90?

The new constructs of Fortran 90 encourage the creation of abstract data types, encapsulation, information hiding, inheritance, generic programming and many features (beyond array syntax), to ensure the safe development of advanced programs. Additionally, the Fortran 90 standard is backward compatible with Fortran 77 allowing new features to be incrementally introduced into existing applications. This is vital as it allows scientific productivity to continue as new concepts and ideas are acquired. Many of the new ideas in Fortran 90 fit in well with object oriented programming [2]. Since Fortran 90 is also a subset of High Performance Fortran, the language forms a migration path to parallel computation using object oriented techniques.

For example, in a particle plasma code, particles (represented by a set of continuous variables), must be moved under the influence of a field, (represented by a discrete grid). In a one-dimensional Fortran 77 implementation we might see the following:

```
dimension part(idimp,np), q(nx), fx(nx)
data qme,dt /-1.,.2/
call push1 (part,fx,qtme,dt,wke,idimp,np,nx)
call dpost1 (part,q,qme,np,idimp,nx)
```

where an array of particles (part), charge density field (q) and electric force field (fx) are used in the operations of pushing particles (push1) and depositing charge (dpost1). Additional parameters include the charge on an electron (qme), the time step (dt), field dimension (nx), particle kinetic energy (wke) and number of particles in the electron species (np). Ideally, it would be preferable to introduce abstractions that simplify the representation of such operations and their components:

```
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)
```

In this Fortran 90 version, we have encapsulated electron and ion species properties into a species derived type, as well as properties of the charge density and electric field. By use association of a module, access to routines and data that define operations on the plasma particles, such as pushing electrons and depositing charge, are now available to the main program. The parameters have been reduced in number and are much clearer. 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.

Encapsulation with Derived Types Encapsulation is critical to object oriented design since this process forms the lowest level abstractions directly under the creation and manipulation of the programmer. Fortran 90 allows users to create their own abstractions from existing intrinsic types and previously defined types using derived types. For instance, a derived type describing a two-dimensional particle can be defined,

```
TYPE particle2d
  real :: x, y, vx, vy ! (x,y) position/velocity
END TYPE particle2d
```

The research of Charles D. Norton was supported by the National Aeronautics and Space Administration Graduate Student Researchers Program under grant NGT-70334, that of Viktor K. Decyk was sponsored by USDOE and NSF and that of Boleslaw K. Szymanski was supported by the NSF under grant CCR-9527151.

where a variable of type **particle2d** can be created using the following syntax:

```
TYPE (particle2d) :: p
```

Components of the particle p can be accessed using the percentage symbol,

```
x_position = p%x ; p%vy = 5.6237
```

where the ';' allows multiple statements to appear on a single line.

As mentioned, derived types may use existing types in their definition. A species type may be created, which consists of a collection of particles with features that describe the species.

```
TYPE species2d
  real :: qm, qbm, ek
  integer :: nop
  integer :: npp
  TYPE (particle2d), DIMENSION (:), POINTER :: p
END TYPE species2d
```

The charge, charge/mass, kinetic energy, total number of species particles and the number on the current processor are encapsulated into the species type. Notice that we have created a pointer to a one-dimensional dynamic array of particles. Fortran 90 provides a number of array types with array operations automatically supported. An **automatic array** is a dynamic array which is created upon entry and destroyed upon exit to a procedure automatically. An **allocatable array** can be created with an **allocate** statement and destroyed with a **deallocate** statement completely under the user's control. Such arrays retain information about themselves, such as their dimensions. Creating multidimensional dynamic arrays is as simple as modifying the parameters to the **dimension** attribute. (Since automatic arrays are not allowed in derived type definitions, a pointer to an allocatable array has been used.) Fortran 90 supports a number of additional important array types.

Abstraction with Modules Oftentimes, derived types are used as parameters to procedures. Fortran 90 requires that the arguments to procedures are type matched, therefore **interface** declarations (similar to function prototypes in C++) are required when derived types are used in this way. These declarations can be provided automatically if derived types are defined within modules; a very important new feature of Fortran 90 that makes its most powerful features available.

A module which defines species properties, with operations on the derived types within the module, appears as follows:

```
MODULE species_module
  USE distribution_module
  USE slab_partition_module
  IMPLICIT NONE
  TYPE particle2d
    PRIVATE
    real :: x, y, vx, vy
  END TYPE particle2d
  TYPE species2d
    real :: qm, qbm, ek
    integer :: nop, npp
    TYPE (particle2d), DIMENSION (:), POINTER :: p
  END TYPE species2d
CONTAINS
  SUBROUTINE spec_dist(species,edges,distf,noff)
    TYPE (species2d), INTENT (out) :: species
    TYPE (slab), INTENT(in) :: edges
    TYPE (distfcn), INTENT (in) :: distf
    INTEGER, INTENT (in) :: noff
    ! subroutine body...
  END SUBROUTINE spc_create
  ! additional routines...
END MODULE species_module
```

The **contains** statement indicates a list of routines, belonging to the module, which are made available where the module is used. A **spec_dist** routine is used to distribute the particle species based on a distribution function (**distf**) and the slab partitioning (**edges**). The **intent** statement adds safety to the interface of the module since we can specify arguments as input-only, output-only or both (**inout**). Also, since we may not want users of the **species_module** to have direct access to the particle components, the **private** attribute restricts access to routines defined within the module, although variables of the derived type can still be created by use association of the module where necessary. In fact, routines can be private to the module if included on a **private** attribute list.

The use statements make the features of existing modules available to the **species_module**. This is not a textual substitution and, as will be seen shortly, this statement is more powerful than the Fortran **include** statement. The **implicit none** statement indicates that all variables must be declared before they are used. Modules allow encapsulation of derived types and the routines that operate with them, which can be "used" wherever module variables are required.

Inheritance with the USE Statement Modules may "use" each other, as long as circular references are not created, providing a mechanism for modeling inheritance. In its simplest form, a base module can make data available to a derived module (shown side by side).

```
MODULE base_module          MODULE derived_module
  IMPLICIT NONE               USE base_module
  INTEGER :: base_data        IMPLICIT NONE
END MODULE base_module        REAL :: derived_data
                            END MODULE derived_module
```

Whenever the derived module is accessible by use association, the base module data is also available.

```
PROGRAM inherit_test
USE derived_module
IMPLICIT NONE
  ! base & derived module data available...
END PROGRAM inherit_test
```

However, we can build more complex derived types and procedures, resembling inheritance in C++, by extending this technique.

For example, given a base module that contains a routine to advance a particle, called **push**, we may want to create a derived module for advancing a collection of particles. Additionally, **push** should be callable by a base or derived module object where the action performed depends on the object type. Assume that the **base_module** created an interface to a **push_one_particle** routine called **push**,

```
  MODULE base_module
    IMPLICIT NONE
    INTERFACE push
      MODULE PROCEDURE push_one_particle
    END INTERFACE
  END MODULE base_module
```

where the details of the module are omitted. A derived module can inherit this push routine to advance a collection of particles in a loop.

```
MODULE derived_module
  USE base_module    ! inherit base_module
  IMPLICIT NONE
  INTERFACE push
    MODULE PROCEDURE push_all_particles
  END INTERFACE
CONTAINS
  SUBROUTINE push_all_particles(this)
  TYPE (particle), DIMENSION(:), & ! continued...
                   INTENT(out) :: this
  INTEGER :: i
    do i = 1, SIZE(this) ! length of array...
       ! calls push_one_particle from base_module
       call push(this(i))
    enddo
  END SUBROUTINE push_all_particles
END MODULE derived_module
```

Since push has been overloaded and the derived module has access to public base module routines, whenever the derived module is used, the proper action will be taken based on the object type:[1]

```
PROGRAM inherit_test
  USE derived_module
  IMPLICIT NONE
  INTEGER, PARAMETER :: nx = 10000
  TYPE (particle) :: x
  TYPE (particle), DIMENSION(nx) :: y
    call push(x)     ! push one particle
    call push(y)     ! push many particles
    stop
END PROGRAM inherit_test
```

Commentary on Inheritance Issues To what extent can Fortran 90 support inheritance? The answer is somewhat controversial, depending on comparison to other languages. For example, a very beneficial feature of C++ is that derived class objects can be used wherever base class objects are expected. In C++, objects created from derived classes are physically modified by inheritance and they acquire all the components of the base class, including typing features. Since Fortran 90 objects are created from derived types (not modules which only make derived types available) and since the use statement does not modify the structure of the derived module, inheritance in Fortran 90 provides functionality, but not the typing features of C++ inheritance. That is, derived objects in Fortran 90 can access base module routines, but the typing restrictions of the language do not allow derived module objects to be used where base module objects are expected. There is no type promotion as in C++; Fortran 90 objects have exactly one type which must be an exact match for all routine arguments in which they are used.

[1]This example only illustrates how an inheritance mechanism similar to C++ may be designed, our codes always push particles as a collection.

Nevertheless, it is still possible to construct derived type objects within the derived module from Fortran 90 base module derived types via inheritance, but the construction must be explicit. (In C++, the compiler implicitly (automatically) includes bass class components in the derived class definition.) Inheritance does not exist as an explicit language feature in Fortran 90, but the statements of the language allow for construction of a useful definition appropriate to this language. Using such definitions to copy inheritance in C++ may lead to an awkward program. However, an advantage of use association of modules with their derived types for inheritance is that a wide variety of Fortran 90 types can participate in the definition of new modules through inheritance. This flexibility allows for beneficial abstract modeling difficult to realize in languages that inherit all features of base classes into derived classes.

We have seen that the use statement allows inheritance of data and routines where interfaces may be created allowing base and derived modules to share routine names. Using similar techniques, new derived types may also be created from existing derived types. Additionally, the use only statement allows module components to be used selectively, hence not everything need be inherited into derived modules.

The Object Oriented Programming Model In object oriented programming, the module has the features of a class while the derived type variables are the objects. In languages like C++, objects are created from classes; they have all the properties of the class definition and routines are bound to the objects. In the Fortran 90 model however, objects are created from derived types by use association of modules. Routines that modules make available can be applied to a single Fortran 90 object, or to a collection of objects. The C++ model can be emulated in Fortran 90 if we restrict a single derived type to each module, however, the ability to include as many derived types as necessary in module definitions opens interesting modeling options not available in other modern languages. The is one example of how Fortran 90 has extended ideas from previous programming models and languages.

2 Plasma Particle-in-Cell Programming

We have been investigating traditional and modern object oriented paradigms in modeling plasma particle-in-cell (PIC) simulations. The PIC model [1] follows the trajectories of millions of particles in their self-consistent electromagnetic fields (both external and self-generated). Since the computation required can be extremely large, both in terms of the field sizes and number of particles, the only alternative is to use massively parallel computers. Computational modeling of plasma using the General Concurrent Particle-in-Cell model [5] involves operations on particles and fields that are partitioned across a distributed memory architecture. Finding appropriate abstractions to represent both the physical and computational/numerical aspects of this problem is challenging due to complex interactions among simulation components.

Object oriented techniques as applied to problems in plasma PIC programming including comparison of performance, modeling, programmability, languages and related issues have been studied in depth [7]. Much of that work addresses comparative issues between Fortran 77, Fortran 90 and C++ for sequential and parallel programming. Although High Performance Fortran (HPF) [4] compilers are

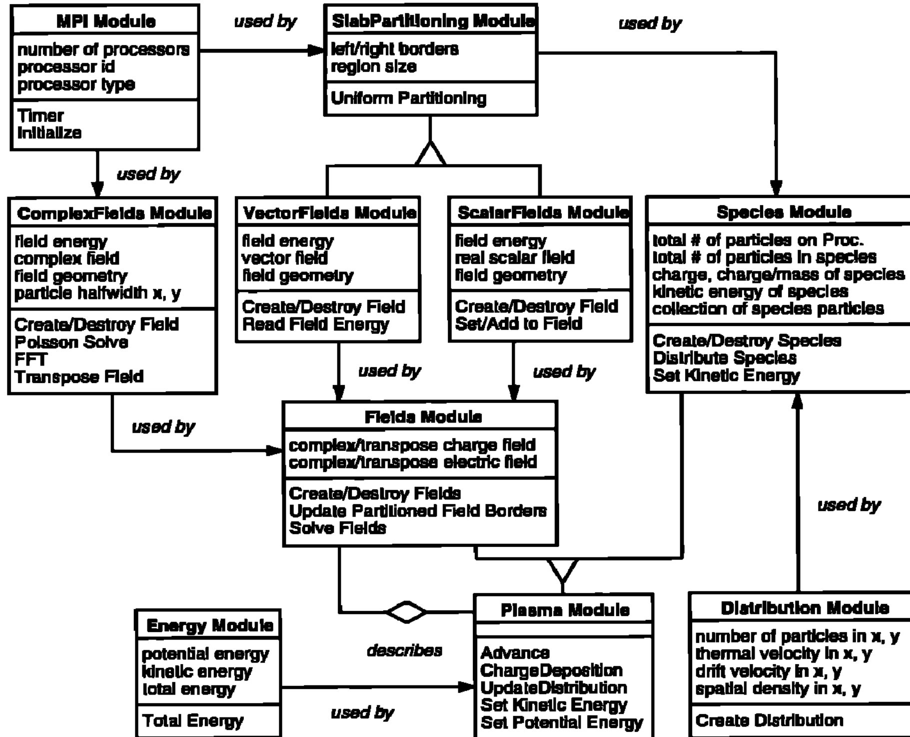

Figure 1: The module hierarchy for a two-dimensional Fortran 90 parallel program, using the OMT notation. Many of the modules contain derived types as part of their definition, such as the definition of a particle in the species module, or a vector field point in the vector fields module.

becoming more widely available, most of the compilers currently available support the subset-HPF standard and some do not yet support important Fortran 90 features such as modules. To learn more about the performance and design of parallel programs in Fortran 90, we extended our sequential model into a parallel class hierarchy which was implemented in Fortran 90 with message passing in MPI. This hierarchy, shown in figure 1, is identical to the sequential model except for the addition of the MPI Module, SlabPartitioning Module and new features added to existing components. (Modifications to the MPI header file were required, consisting mainly of interface declarations for routines, since the Fortran 77 library calls were used in Fortran 90.)

The Object Modeling Technique [9] (OMT) notation illustrates modules with their primary data attributes (components) followed by the major routines that operate on the data. The links among modules show various relationships. The VectorFields and ScalarFields modules inherit from the SlabPartitioning module, allowing the definition of these fields to include partitioning across processors. The Plasma module inherits from both the Species (indicating the particle collection) and the Fields module since the definition of the plasma consists of these components. In fact, the plasma module contains no data, it only provides functionality for fundamental operations. The annotations on the directional links indicate how use association provides information between modules. The electron species for instance, must know about the simulation space partitioning across processors as well as the distribution function used in initializing particles. This information can be made available by use association, as seen in the Abstraction with Modules section. Various particle species can be created.

The ComplexFields module computes numerical operations on fields represented in complex space. The Fields module provides routines for managing the creation and destruction of fields and communication operations involving guard regions required for efficient message passing of field components. Finally, the Energy diagnostic computes the total energy of the system from the field and particle energy components.

Machine	Fortran 77	Fortran 90	C++
IBM SP2	195.08 (sec)	202.88 (sec)	359.00 (sec)

Table 1: Performance characteristics for 2D two-stream instability experiment on IBM SP2 (AIX 4.1) with 3.5 million particles on 32 processors.

Performance of this code, compared to the original Fortran 77 and a C++ version are shown in table 1 for the IBM SP2 using the xlf, xlf90 and xlC compilers. A sketch of the main program segment is shown in figure 2, where object creation and usage in Fortran 90 is illustrated. The abstractions are clearly defined, and as seen in table 1, the performance is competitive; but this varies based on the compiler used [7].

Routines to distribute particles, deposit charge, solve for the electric field, advance particles and redistribute them across processors use the abstractions created from derived types within modules. Since the partition module uses the MPI module, all MPI functions are also known to the main program. Various routines use the Fortran 90 generic function mechanism which allows routines to share the same name, but respond differently based on the argument type

```
PROGRAM beps2k
  USE partition_module, plasma_module
  TYPE (distfcn) :: backdf, beamdf
  TYPE (species2d) :: electrons
  TYPE (sfields2d) :: cdensity
  TYPE (vfields2d) :: efield
  TYPE (energy) :: energ
  TYPE (slab) :: edges
  call MPI_INIT(ierror)
  ! distribute background and beam electrons
  call species_dist(electrons,edges,backdf,noff)
  call species_dist(electrons,edges,beamdf,nps)
  ! initial charge deposition
  call plasma_dpost(electrons,cdensity,edges)
  DO itime=1,500
    ! calculate force/charge
    call fields_solve(cdensity,efield)
    ! push particles and move across PEs
    call plasma_push(electrons,efield,edges,dt)
    call plasma_pmove(electrons,edges)
    call plasma_dpost(electrons,cdensity,edges)
    ! add background ion density
    call sfields_add(cdensity,edges,qi0)
    ! energy diagnostic
    call energy_gette(energ)
  END DO
  ! destroy dynamic structures
  call fields_destroy(efield)
  call fields_destroy(cdensity)
  call species_destroy(electrons)
  MPI_FINALIZE(ierror)
  stop
  end
END PROGRAM beps2k
```

Figure 2: Sketch of the main program from the two-dimensional Fortran 90 parallel code.

(as specified via interface blocks). Function resolution is performed at compile-time, so run-time polymorphism in languages such as C++ is not provided, for reasons also related to typing issues in Fortran 90.

3 Summary

While Fortran 90 may not be considered an object oriented language by most definitions, many of its new features support object oriented concepts. This can simplify the process of extending existing programs into a parallel environment since modifications to encapsulated components can be introduced in a safe manner without unwanted side-effects. These techniques may benefit Fortran 77 programmers looking to use the object paradigm in their existing scientific programs. While languages such as C++ also have many advanced and useful features for sophisticated programming, such as templates, in general Fortran 90 provides an easier transition to object modeling and programming for Fortran programmers interested in the benefits of this methodology. This is true because Fortran 90 is still geared toward scientific programming, not necessarily general purpose programming. Fortran 90 also contains advanced features, including the array syntax notation, pointers and a clearly defined way to create multidimensional dynamic structures, to name a few.

The object oriented methodology should not be confused with its representation by a specific language since the definition of object oriented programming is language independent. Many of the features of Fortran 90 are beneficial even if the object oriented approach is not fully adopted. Nevertheless, the methodology can establish a context through which these new features can be applied. While it is possible to mimic aspects of C++ in Fortran 90, this can be tedious and inappropriate at times. In [10], it is suggested that languages should support a style of programming based on the ease, convenience and safety by which the features of that style are supported. Excessive effort in modeling a style of programming indicates that the language simply enables various techniques to be applied. We believe that judicious use of modules, use association and derived types in abstraction modeling (with other advanced features) will lead to a programming style appropriate to the semantics of Fortran 90, not necessarily other languages. This philosophy has been very successful in our sequential and parallel application development. The reader is encouraged to explore the references for additional details beyond what has been presented here.

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] V. K. Decyk, C. D. Norton, and B. K. Szymanski. Introduction to Object-Oriented Concepts Using Fortran 90. Technical Report PPG–1559, Institute of Plasma and Fusion Research, UCLA Dept. of Physics and Astronomy, Los Angeles, CA 90095-1547, June 1996. Submitted to Computers in Physics.

[3] T. M. R. Ellis, I. R. Philips, and T. M. Lahey. *Fortran 90 Programming.* Addison–Wesley, Reading, M.A., 1994.

[4] High Performance Fortran Forum. *High Performance Fortran Language Specification,* version 1.0 edition, May 1993. Technical Report CRPC-TR92225, Rice University, Houston, January 1993.

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

[6] S. B. Lippman. *C++ Primer.* Addison–Wesley, Reading, MA, second edition, 1991.

[7] C. D. Norton. *Object Oriented Programming Paradigms in Scientific Computing.* PhD thesis, Renssleaer Polytechnic Institute, Troy, New York, August 1996.

[8] C. D. Norton, B. K. Szymanski, and V. K. Decyk. Object Oriented Parallel Computation for Plasma Simulation. *Communications of the ACM,* 38(10):88–100, October 1995.

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

[10] B. Stroustrup. *The C++ Programming Language.* Addison–Wesley, Reading, MA, second edition, 1991.