IMPLEMENTATION OF OBJECT-ORIENTED DESIGN WITH FORTRAN LANGUAGE IN BEAM DYNAMICS STUDIES

J. Qiang, R. Ryne, and S. Habib R. Ryne, LANL, Los Alamos, NM 87545 USA

Abstract

In this paper, an object-oriented design for beam dynamics simulations in accelerators is implemented using Fortran language. Using module and derived type in F90, we can emulate object concept in the object-oriented design. This gives Fortran code a better maintainability, reusability, and extensibility. The overhead associated with the object-oriented implementation has only a minor effect on performance.

1 INTRODUCTION

Object-oriented design is being widely applied in computer software engineering to implement complex codes which possess good maintainability, reusability, and extensibility. This technique also enables the encapsulation of detailed machine specific information, thereby achieving good portability.

In the parallel computing environment, such efforts have mostly been directed to the design of object-oriented frameworks using explicit message passing and C++ [1]. Using such an object-oriented framework reduces the extent of difficulty of parallel programming based on message passing library and also allows good performance to be achieved. However, in the physics community, e.g., the accelerator community, Fortran still remains a popular language for demanding numerical simulations. Even here, implementation of object-oriented design can be useful since using F90 with Message Passing Interface (MPI) in this way encapsulates the detailed communication syntax and eases the design and implementation of parallel simulations.

In contrast to the message passing paradigm discussed above, High Performance Fortran (HPF) as a high level data parallel programming language also has its place in scientific computation. Its advantages of programming ease, reasonable performance, and portability between parallel and serial machines makes it attractive for use in many applications [2, 3, 4]. In HPF, inter-processor communication is handled by the compiler. The programmer generally only needs to explicitly specify the data distribution on parallel processors and parallel loops through directives comments [5]. This makes parallel programming more transparent and allows portability between parallel and serial machines.

Though not designed with object-orientation in view, Fortran 90 already contains some features of object-oriented programming languages with user defined generic data type, pointer, and modules [6, 7]. These features have been successfully applied in plasma simulations to build object-oriented Fortran codes [8]. Since HPF is supposed to contain all the features of Fortran 90, it is also possible in theory to emulate object-oriented programming using the intrinsic module command and derived types in HPF. Thus, the application of object-oriented design with HPF can combine the traditional advantages of object-oriented methods along with the ease of parallel programming that characterizes HPF.

In this paper: The physical system is described in Section 2, object-oriented design is presented in Section 3, parallel implementation using F90 with MPI is given in Section 4, data parallel implementation with HPF is discussed in Section 5, numerical results on the SGI/Cray T3E-900 and SGI Origin2000 are presented in Section 6, and the conclusions summarized in Section 7.

2 PHYSICAL SYSTEM

The physical system for beam dynamics studies consists of the beam and the accelerating system which in turn contains a number of accelerating, guiding, and focusing elements. The forces acting on particles are due to externally applied fields and the inter-particle Coulomb field.

The two-dimensional application we will consider below is a study of the transverse dynamics of an infinitely long intense beam transporting across various focusing elements along the z-axis. We note that since accelerating, guiding, and focusing elements are arranged along z, it is usual practice in accelerator simulations to take z to be the independent variable rather than the time t. In this case, the original six dimensional equations reduce to a set of four dimensional z-dependent equations

$$\frac{\partial f}{\partial z} + x'\frac{\partial f}{\partial x} + y'\frac{\partial f}{\partial y} + v_x'\frac{\partial f}{\partial x'} + v_y'\frac{\partial f}{\partial y'} = 0 \quad (1)$$

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = \rho/\epsilon \quad (2)$$

Here, f is the particle distribution function in phase space, ϕ is the space charge potential, ρ is the charge density from the distribution function, a prime superscript denotes a derivative with respect to z. The above equations is solved using a particle-based method. Particle simulations have much lower storage cost (in three spatial dimensions, N^3 vs. N^6) and have the crucial advantage of not breaking down when phase space structure falls below the grid resolution.

In the case of a linear focusing system, the Hamiltonian for single particle dynamics is

$$H = \frac{1}{2m}(p_x^2 + p_y^2) + H_{ext} + H_{self} \quad (3)$$

where m is the mass of the particle, p is the momentum, H_{ext} is the external field contribution, and H_{self} is the space charge contribution. Numerical solution of the particle equation of motion is easy to obtain using a split-operator sympletic integration method. First, one breaks up the above Hamiltonian into two parts, i.e. $H = H_1 + H_2$, where it is assumed that particle motion under either one of H_1 or H_2 can be solved for exactly. In that case, a second order integrator is defined by the map

$$\mathcal{M}(\tau) = \mathcal{M}_1(\tau/2)\mathcal{M}_2(\tau)\mathcal{M}_1(\tau/2) \qquad (4)$$

where $\mathcal{M}$ represents a map to integrate the particle equation of motion from one state to another state in phase space, $\mathcal{M}_1$ is the map corresponding to H_1, $\mathcal{M}_2$ is the map corresponding to H_2, and τ is the step size [9].

Often, the beam size is much smaller than the inside wall radius of the accelerator, in which case we may treat the beam as an isolated system. This results in open boundary conditions for the Poisson equation which can then be solved by a Fast Fourier Transform (FFT) technique using a method given by Hockney [10]. While the beam moves through the accelerator, its cross section varies along the axis. The computational grid used for solving the Poisson equation has therefore to be adjusted in size in order to follow the beam with a constant transverse resolution.

3 OBJECT-ORIENTED DESIGN

Object-oriented design is a method of design encompassing the process of object-oriented decomposition [11]. During the process of object-oriented design, the complexity of the system is decomposed into a number of objects which are instantiated from their corresponding classes. An object, as defined by Booch, is a tangible entity which exhibits some well defined behaviors, and a class is a set of objects that share a common structure and a common behavior. There are four fundamental elements contained in the concept of an object-oriented model. These are abstraction, encapsulation, modularity and hierarchy.

Abstraction is a process that extracts the essential common characteristics of a set of objects that distinguish it from other sets of objects. It provides an outside view of an object and defines a conceptual boundary of the object. Encapsulation is also called "information hiding." It is complementary to abstraction and screens from outside viewers all inside details of an object that do not contribute to its essential characteristics. Modularity is a property used to decompose a system into a set of meaningful, cohesive, and loosely coupled modules. Each module consists of a number of classes and objects working together to model specific aspects of system behavior. Hierarchy establishes the inter-relationships among classes in an object-oriented model by ranking or ordering the abstractions. The two most important relationships of class are inheritance and aggregation. The inheritance specifies a generalization/specialization hierarchy, i.e. "kind of" hierarchy. The aggregation specifies a containing hierarchy, i.e. "part of" hierarchy.

In objected-oriented design, after analysis of the (complex) physical system, the system is first decomposed into simpler physical modules. Next, objects are identified inside each module. Then, classes are abstracted from these objects. Each class has interfaces to communicate with the outside environment. Then relationships are built up among different classes and objects. These classes and objects are implemented in a concrete language representation. The implemented classes and objects are tested separately and then put into the physical module. Each module is tested separately before it is assembled into the whole program. Finally, the whole program is tested to meet the requirements of problem.

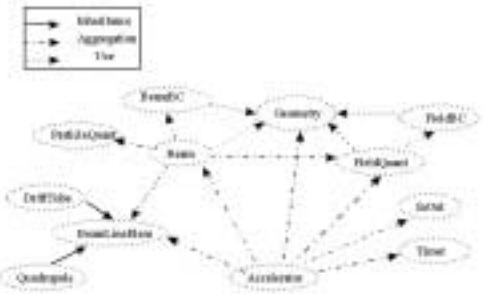

Figure 1: The class diagram of the accelerator beam dynamics system.

An application of the object-oriented design methodology outlined above to beam dynamics studies in accelerators results in the decomposition of the physical system into five modules. The first module handles the particle information consisting of Beam and BeamBC classes. The second module handles information regarding quantities defined on the field grid, and contains the FieldQuant and FieldBC classes. The third module handles the external focusing and accelerating information containing the BeamLineElem base class and its derived classes. The fourth module handles the computational domain geometry containing Geometry class. The last module provides auxiliary functions containing InOut and Timer classes. The class diagram of the object-oriented model for a beam dynamics study is presented in Fig. 1. In this class diagram, we do not include the auxiliary classes used in the parallel implementation using MPI. These classes are implemented using both F90 with MPI and HPF in our parallel beam dynamics simulations.

4 PARALLEL IMPLEMENTATION WITH F90 AND MPI

MPI is a standard library of message passing parallel programming bound to C and F77 [12]. It provides a direct access to the physical architecture. The programmer has to control the data distribution on the processors and commu-

nications among processors when information from more than one processor is required. This gives it the advantages of flexibility and better performance. However, this also increases the difficulty of parallel programming. Applying object-oriented design to parallel message passing programming helps to encapsulate the details of communications and data distributions. This enables the user to manage the applications at a higher level.

To implement the object-oriented design with F90 and MPI, we add some new classes to the original auxiliary module to handle explicit communications. These classes are Pgrid2d, Communication, and Utility. The Pgrid2d class defines a logical two dimension Cartesian processor grid. The Communication class contains the public functions to handle the major communications used in the particle-in-cell simulation using MPI. The Utility class contains three public functions to encapsulate the explicit communications used in the general purpose function operations, e.g. matrix transpose. With the help of auxiliary class module, the particle simulation using beam, field, beam line element, and geometry modules can be built up without knowing the details of the communications. In the following, we give an example of implementing the F90 to emulate polymorphism in the beam line elements in our simulation. The polymorphism is done in an object-oriented language by defining a virtual base class and different derived classes. By assigning the address of a derived class object to a pointer object of base class, the procedure using a single base object name can select the appropriate member function to execute based on the actual class object referenced in the pointer object. In our beam dynamics simulation with F90, we define a base class BeamLineEment, and three derived classes for the drift, focusing, and defocusing beam line elements. The scaled down sketch of these classes are below:

```
module BeamLineElemclass
  use QuadrupoleFclass
  use QuadrupoleDclass
  use DriftTubeclass
  type BeamLineElem
    private
    type (QuadrupoleF), pointer :: pquadf
    type (QuadrupoleD), pointer :: pquadd
    type (DriftTube), pointer :: pdrift
  end type BeamLineElem
  interface assign_BeamLineElem
    module procedure assign_quadf,
           assign_quadd,assign_drift
  end interface
contains
function assign_quadf(pquadf) result(ppquadf)
function assign_quadd(pquadd) result(ppquadd)
function assign_drift(tdrift) result(ppdrift)
subroutine update_BeamLineElem(this,z0,z1)
end module BeamLineElemclass

module DriftTubeclass
  integer, private, parameter :: Nparam = 1
  type DriftTube
    integer :: Nseg
    real :: Length
    real, dimension(Nparam) :: Param
  end type DriftTube
contains
  subroutine update_DriftTube(this,z0,z1)
end module DriftTubeclass
```

Here, only the drift tube class is given for the derived class since the other two derived classes have a similar structure to the drift tube class. Since there is no direct support of inheritance in F90, we define a derived type in the BeamLineElem base class which contains three pointers to the derived classes as private data members. An overloaded function assign_BeamLineElem which includes three assignment functions is used to initialize the base BeamLineElem class object with different derived class object addresses. In each assignment function, only one pointer is initialized and the other two pointers are set to null. In the public function update_BeamLineElem of the base class, updating operations from derived classes are selected according to the different actual object association of pointers in the base class data member. The polymorphism is achieved by calling this subroutine with a constructed base BeamLineElem object in the application. The data members in the derived beam line element class consist of the number of steps of particle movement inside the beam line, length, and strength of the beam line element.

5 DATA PARALLEL IMPLEMENTATION WITH HPF

Data parallel programming using HPF provides a relatively easy route to parallel programming. Present compilers, however, are still not fully mature and performance penalties are often encountered.

In principle, using derived type with private data member in an HPF module containing some public member functions, it is possible to emulate a class in the same way as F90 does. Unfortunately, most present HPF compilers do not have adequate support for derived type and dynamically distributed arrays. For example, the PGHPF compiler does not support pointer to derived type, deferred array component in derived type, and parallel distribution of array component to processors in derived type [13]. These restrictions prevent one from defining a generic derived type with dynamic array component inside a module. Emulating object-oriented polymorphism is not possible for the same reason. Therefore, to implement the object-oriented design discussed in the Section 3, we have to modify some classes in our implementation from a generic type to a physical module which contains some private data members and public member functions. The public member functions contained in the physical module provide the behaviors of the module.

In the previous section, we showed that polymorphism could be used in the implementation of beam line elements. However, due to the absence of support for a pointer to derived type in the PGHPF compiler, we have to include the choice of different concrete beam line elements in the BeamlineElem module as separate subroutines. A scaled

down sketch of beam line element module is presented in the following.

```
module BeamLineElem_class
  integer, private, parameter :: Nparam = 1
  type BeamLineElem
    private
    integer :: Nseg
    real :: Length
    real, dimension(Nparam) :: Param
  end type BeamLineElem
contains
  subroutine update_BeamLineElem(this,flag,
                                 z0,z1)
  subroutine beamlnDeQuad(this,z0,z1)
  subroutine beamlnFoQuad(this,z0,z1)
  subroutine beamlndefault(this,z0,z1)
end module BeamLineElem_class
```

Parallel loop implementation uses the HPF commands *Do Independent* and *Forall*. *Forall* is used in the case of a single statement with regular array index access. *Do Independent* is used to fuse several statements into one loop to take advantage of the spatial and temporal locality of data in cache. (In the data scattering from grid to particle subroutine, we observed that using *Do Independent* is about a factor of ten faster than using *Forall* in the indirect array index access loop.)

6 NUMERICAL RESULTS

The above object-based Fortran programs were applied to the study of proton beam transport through a periodic constant focusing, drift, defocusing, drift (FODO) channel. Here, we report our experience running on the SGI/Cray T3E-900 and SGI Origin2000.

The problem we tested here consists of 10 FODO periods. Our simulation used $250,000$ particles with a 128×128 field mesh grids. Fig 2 gives the performance of using the object-based F90 with MPI on the SGI/Cray T3E-900.

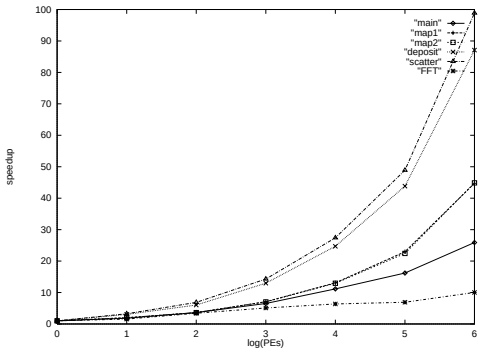

Figure 2: The speedup of the object-based F90/MPI code as a function of the number PEs on SGI/Cray T3E-900.

We measured the speedup of 5 major subroutines scaled by the number of processors. These are map1, map2, scattering, depositing and FFT subroutines. The speedup is calculated as the ratio of time measured on one processor to the time measured on a given number of processors. The main program speedup reaches about 26 on 64 processors. The total efficiency (ratio of speedup to number of processors) here is relatively low due to the heavy communication costs and small problem size. Checking the performance of separate subroutines, we see that the FFT subroutine is the least scalable due to the global nature of the Fourier transform resulting in communication overhead. On the other hand, depositing and scattering subroutines show a superlinear trend which give a speedup close to 100 on 64 processors. This is because, in the F90/MPI code, since both subroutines are done locally on each processor, the resulting more efficient use of cache may provide superlinear speedup in the operation of these subroutines. The performance of the corresponding HPF code is given in Fig 3.

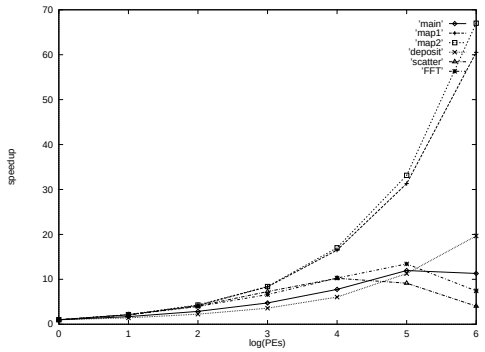

Figure 3: The speedup of the object-based HPF code as a function of the number PEs on SGI/Cray T3E-900.

We see that the main program speedup increases to 32 processors and saturates beyond that. Checking the speedup on individual subroutines, we find that the grid-particle scatter subroutine reaches its maximum speedup on 16 processors due to heavy communication in the indirect array index access. This becomes a bottleneck for the case of a large number of processors. The FFT subroutine reaches its maximum speedup on 32 processors. The map1 and map2 subroutines scale very well with increasing number of processors. The depositing subroutine does not scale well due to the communications in particle deposition. Fig 4 gives the speedup performance of the same code on the SGI Origin2000.

The main program speedup increases to 32 processors and starts to decrease on 64 processors. The speedups of individual subroutines map2, and the depositing and scattering subroutines are superlinear which might be due to local cache effects. The FFT subroutine gives very poor scalability saturating even on 16 processors due to its global nature. This makes the main program less scalable on the SGI Origin2000 than on the SGI/Cray T3E-900. To implement the object-oriented design, we used a number of pointers and dynamics allocated arrays. This will affect the code performance due to loss of compiler optimization comparing with non-object-oriented code. In Fig 5, we give a comparison of the time costs of object based codes using both F90/MPI and HPF with conventional procedure based codes.

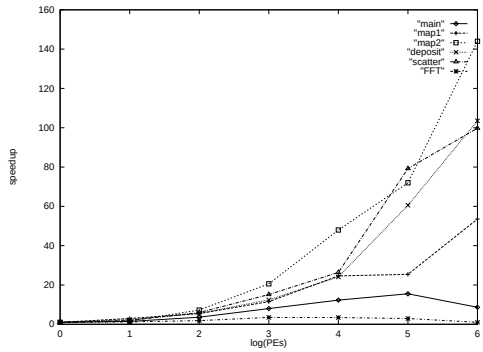

Figure 4: The speedup of the object-based F90/MPI code as a function of the number PEs on SGI Origin2000.

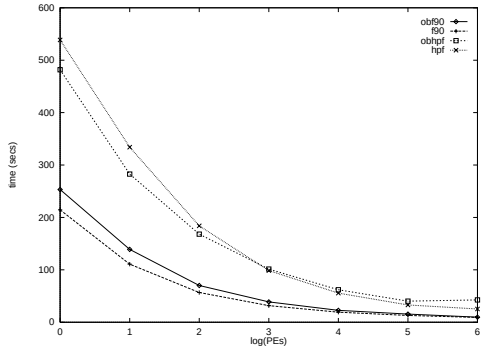

Figure 5: The time costs of object-based and procedure-based programs as a function of the number PEs on SGI/Cray T3E-900.

For F90 with MPI code, on small number of processors, the overhead from object-oriented implementation is about 10 to 20%. This overhead decreases with increasing number of processors. In the case of HPF, the object based code seems to over perform the procedure based code on small number of processors and lose out on a large number of processors.

7 CONCLUSIONS

In this paper we have discussed implementations of object-oriented design using Fortran in parallel beam dynamics simulations. As previously stated, implementing the object-oriented design with F90 and MPI encapsulates the details of communication in low level auxiliary classes. This also provides the benefits of better maintainability, reusability and extensibility of software. For example, a new beam line element can be easily incorporated into the BeamLineElem class without affecting the other classes. Using the concept of object, which is implemented using HPF modules, gives the code some advantages of object-oriented programs and also the advantage of ease in parallel programming. This implementation was based on the current status of the PGHPF compiler technology. With further development of compilers, it is possible that the programming barriers we encountered will disappear. In that event, this model will be easily extended to adopt new features in the future to include more completely object-oriented features.

In our first experience of applying these codes on the SGI/Cray T3E-900 and SGI Origin2000, we obtained a reasonable performance up to 32 processors on both machines. The code written using F90 with MPI seems to be more scalable on the SGI/Cray T3E-900 than on the SGI Origin2000. The overhead of implementing object-oriented design using pointers, types and modules is small. To summarize, it appears to us that implementing object-oriented design with Fortran can achieve both good software quality and parallel programming in scientific applications.

8 ACKNOWLEDGMENTS

We acknowledge helpful and stimulating discussions with Dr. Viktor Decyk and the POOMA team, who provided POOMA beam dynamics source code. This work was performed on the SGI/Cray T3E at NERSC and SGI Origin2000 at the ACL and supported by the DOE Grand Challenge in Computational Accelerator Physics.

9 REFERENCES

[1] G. Wilson, L. Paul (ed.): Parallel Programming Using C++, MIT Press, Cambridge (1996).

[2] R. Ryne, S. Habib, Parallel Beam Dynamics Calculations on High Performance Computers, In: J. J. Bisognano, A. A. Mondelli (eds.): Computational Accelerator Physics, AIP Conference Proceedings 391, Woodbury, New York (1997) 377-389.

[3] C. H. Ding: HPF for Practical Scientific Algorithms, Preprint for Supercomputing '97, (1997).

[4] V. V. Elisseev, Parallelization of Three-dimensional Spectral Laser-Plasma Interaction Code Using High Performance Fortran, Computers in Physics **12** (1998) 173-180.

[5] C. H. Koelbel, B. D. Loveman, R. S. Schreiber, G. L. Steele, M. E. Zosel : The High Performance Fortran Handbook, MIT press, Cambridge, (1994).

[6] T. M. Ellis, I. R. Philips, T. M. Lahey: Fortran 90 Programming, Addison-Wesley, Harlow, England (1994).

[7] M. G. Gray, R. M. Roberts: Object-based Programming in Fortran90, Computers in Physics **11** (1997) 355-361.

[8] V. K. Decyk, C. D. Norton, B. K. Szymanski : Expressing Object-Oriented Concepts in Fortran 90, ACM Fortran Forum, Vol. 16, num 1, (1997).

[9] E. Forest and R. D. Ruth: Fourth-Order Symplectic Integration, Physica D **43** (1990) 105-117.

[10] R. W. Hockney and J. W. Eastwood: Computer Simulation Using Particles, Adam Hilger, New York, (1988).

[11] G. Booch: Object-Oriented Analysis and Design with Applications, Benjamin/Cummings, Menlo Park, CA, (1994).

[12] http://www-c.mcs.anl.gov/Projects/mpi/ (1998).

[13] PGHPF manual, http://www.nersc.gov/software/prgenv/compilers/pghpf/docs/, (1998).