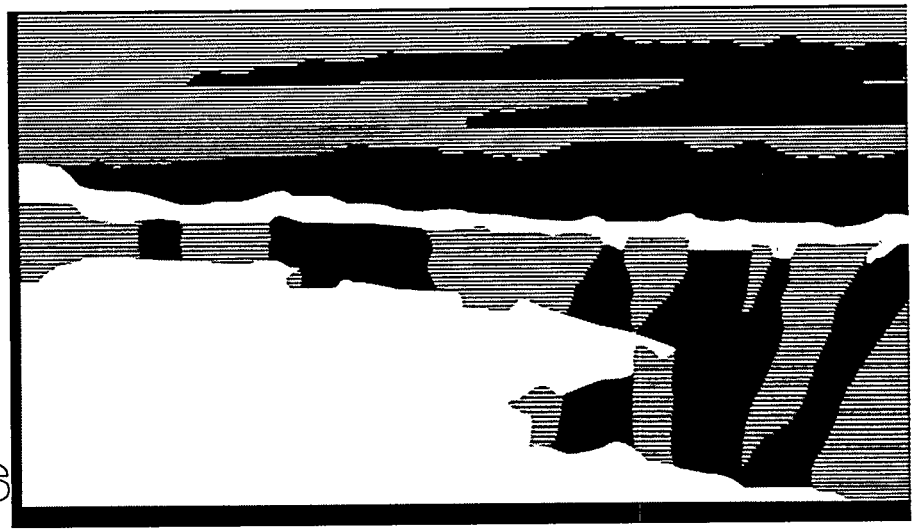
LA-UR-98-3594

TITLE: OBJECT-ORIENTED ACCELERATOR DESIGN WITH HPF

CONF-981207--

AUTHOR(S):
Ji Qiang — LANSCE-1
Robert D. Ryne — LANSCE-1
Salman Habib — T-8

SUBMITTED TO:
ISCOPE '98
SANTA FE, NM
DECEMBER 8-11,1998



Los Alamos
NATIONAL LABORATORY

Form No. 836 R5
ST 2629 10/91

DISCLAIMER

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

Object-Oriented Accelerator Design with HPF

Ji Qiang, Robert D. Ryne, and Salman Habib

Los Alamos National Laboratory
Los Alamos, NM 87545

Abstract. In this paper, object-oriented design is applied to codes for beam dynamics simulations in accelerators using High Performance Fortran (HPF). This results in good maintainability, reusability, and extensibility of software, combined with the ease of parallel programming provided by HPF.

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 [1]. For example, the Parallel Object-Oriented Methods and Applications (POOMA) framework using C++ and message passing has been applied to the study of plasma turbulence, beam dynamics, Monte Carlo neutron transport, etc. [2] Using such an object-oriented framework reduces the extent of difficulty of parallel programming based on message passing and also allows good performance to be achieved. On the other hand, High Performance Fortran as a high level data parallel programming language is also employed in scientific computation due to its advantages of programming ease, reasonable performance, and portability between parallel and serial machines [3–5]. 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 [6]. This makes parallel programming more transparent and allows portability between parallel and serial machines.

Though not designed with object-orientation in view, Fortran is not incompatible with the application of object-oriented methodologies in scientific computation. In fact, Fortran 90 already contains some features of object-oriented programming languages [7, 8] which have been successfully applied in plasma simulations [9]. Since HPF contains all the features of Fortran 90, it is 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. Unfortunately, at present, HPF compilers do not have adequate support for some features

important in object-oriented applications, e.g. derived type with dynamic array. Hence, it is already clear that compromises have to be made in some instances in order to keep the object-oriented design and HPF programming implementation.

In this paper, we present a first attempt to apply object-oriented design with HPF in the simulation of charged particle beams in accelerators. The paper is organized as follows: The physical system is described in Section 2, object-oriented design is presented in Section 3, data parallel implementation with HPF is discussed in Section 4, application on the SGI/Cray T3E-900 is given in Section 5, and Section 6 summarizes the conclusions.

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.

We will consider the case of a beam of particles, the dynamics of which is governed by the Vlasov-Poisson system of equations:

$$\frac{\partial f}{\partial t} + \mathbf{v}\nabla_{\mathbf{r}} f + \mathbf{a}\nabla_{\mathbf{v}} f = 0 \tag{1}$$

$$\nabla^2 \phi = -\rho/\epsilon \tag{2}$$

where f is the particle distribution function in phase space, $\mathbf{v}$ is the velocity, $\mathbf{r}$ is the displacement, $\mathbf{a}$ is the acceleration depending on the external force and space charge potential ϕ, ρ is the charge density from the distribution function, and ϵ is the dielectric constant in vacuum. In three spatial dimensions, the above equations represent a self-consistent initial value problem for a six-dimensional partial differential equation. Direct grid-based simulation of these equations is therefore prohibitively expensive in three spatial dimensions. When symmetries are present the dimensionality is lowered and direct simulations are possible in effectively one and two dimensional situations [3]. However, even in these cases memory restrictions are severe and there are problems with code breakdown when very small-scale structure forms in the phase space (the details of which, in any case, are largely irrelevant). Alternatively, the Vlasov-Poisson equations may be 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.

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

$$\frac{\partial f}{\partial z} + x'\frac{\partial f}{\partial x} + y'\frac{\partial f}{\partial y} + v'_x\frac{\partial f}{\partial x'} + v'_y\frac{\partial f}{\partial y'} = 0 \tag{3}$$

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = \rho/\epsilon \tag{4}$$

Here, a prime superscript denotes a derivative with respect to z.

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

$$H = \frac{1}{2m}(p_x^2 + p_y^2) + H_{ext} + H_{self} \tag{5}$$

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

$$\mathcal{M}(\tau) = \mathcal{M}_1(\tau/2)\mathcal{M}_2(\tau)\mathcal{M}_1(\tau/2) \tag{6}$$

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 [10].

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 [11]. 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.

Computationally, beam transport through the accelerator is performed using the following steps: First, the system is initialized. Next, particles move a half step via free particle motion. Then, geometry, beam line elments, beam boundary conditions and field boundary conditions are updated. The beam charge density on the numerical field grid is then calculated from the deposition of particles. The potential field is updated with the charge density as source. The particles are then kicked in momentum space. Finally, particles free stream another half step. This process is repeated for many times until the particles move out of the accelerator.

3 Objected-Oriented Design

Object-oriented design is a method of design encompassing the process of object-oriented decomposition [12]. 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. Here, an object is defined by Booch as a tangible entity which exhibits some well defined behavior, 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 to extract 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 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.

Application of the object-oriented design outlined above to beam dynamics in accelerators results in the decomposition of the physical system into five modules. The first module handles the particle information consisting of Beam, ParticleQuant 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 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.

Beam class is a major class in the study of beam dynamics. It contains global particle information including beam current, single particle mass and charge, and individual particle position and momentum in one array. It also contains the beam charge density object (defined on a grid) from the instance of FieldQuant class and the force object on particles from the instance of ParticleQuant class. Beam class has three major functions: (1) moving particles in phase space at each time step, (2) depositing particles on the grid to get the charge den-

sity, and (3)scattering a quantity defined on the grid back onto the particles. Beam class uses BeamLineElem, BeamBC, Geometry and Timer classes in its implementation and interface. ParticleQuant class provides information related to individual particles, e.g. force on a particle. BeamBC class provides information related to the particle boundary conditions, e.g., to record which particle is out of the computational domain.

Fig. 1. The class diagram in beam dynamics study

FieldQuant class is another important class. It contains information of quantities defined on field grid, e.g. the potential ϕ. One major function in FieldQuant is to update the quantity on the field grid with an input source. FieldQuant uses FieldBC and Geometry classes. FieldBC contains the information about various boundary conditions on the field quantity.

Geometry class contains the practical and numerical geometry information of the physical system. Its major functions include evaluating a spatial point outside the computational domain, determining boundary grid points, and updating the computational boundary during system evolution.

BeamLineElem class contains the external focusing and accelerating information. It is a virtual base class. The different concrete beam line element classes, e.g. Quadrupole and DriftTube classes, are derived from this class. Polymorphism is used to access and update concrete elements.

Timer class as its major function counts the time spent in each part of the computation. InOut class has two major functions. One to bring input information to the program, and the other to print output information to files.

Accelerator class is a container class. It contains objects instantiated from Beam, BeamLineElem, Geometry, FieldQuant, InOut and Timer classes. Its major function is to run the beam dynamics simulation inside the accelerator.

4 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 object-oriented concept). 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 last section, we have to modify some classes in our object-oriented design from a generic type to a local 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.

The following gives a scaled down sketch of the Beam class module.

```
module Beam_class
  use Geometry_class
  use BeamBC_class
  use BeamLineElem_class
  real, private :: Current, Kinenergy, Mass, Charge
  integer, private :: Ndim, Np, Nx, Ny
  real, private, allocatable, dimension(:,:) :: ParticlesOne
  real, private, allocatable, dimension(:) :: ParticleForceX
  real, private, allocatable, dimension(:) :: ParticleForceY
  real, private, allocatable, dimension(:,:) :: ChargeDensity
!hpf$   distribute ParticlesOne(*,block)
!hpf$   align (:) with ParticlesOne(*,:) :: ParticleForceX
!hpf$   align (:) with ParticlesOne(*,:) :: ParticleForceY
!hpf$   distribute ChargeDensity(*,block)
  interface map1_Beam
  module procedure drift1_Beam, drift2_Beam
  end interface
  interface map2_Beam
    module procedure kick1_Beam, kick2_Beam
  end interface
contains
  subroutine deposit_Beam(innp,innx,inny,rays,rho,msk,ptsgeom)
  subroutine scatter_Beam(innp,innx,inny,rays,ex,exg,msk,ptsgeom)
end module Beam_class
```

Here, only data members and major member function interfaces are shown. The Beam class module contains objects from geometry and beam line elements classes. The particles are represented by a two-dimensional array with each column denoting one particle. Instead of using objects from ParticleQuant with array component which is not supported by PGHPF, we use arrays to represent the forces on particles. The charge density is also represented by a two-dimensional array instead of an object from FieldQuant class for the same reason. These arrays are distributed uniformly onto processors to achieve a good load balance.

In our class diagram, 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 element in the BeamlineElem module as a separate subroutine. 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
```

Quantities defined on a field grid inside the accelerator (e.g., space charge potential) are represented by a two-dimensional array as a private data member of the FieldQuant module instead as a component array in the derived type. This array is distributed blockwise among processors. The scaled down sketch of the FieldQuant module is given below.

```
module FieldQuant_class
  use Geometry_class
  use FieldBC_class
  integer, private :: Nx, Ny
  real, private, allocatable, dimension(:,:) :: FieldQ
!hpf$   distribute FieldQ(*,block)
contains
  subroutine update_FieldQuant(source,fldgeom)
  subroutine fft2d_FieldQuant(n1, n2, ksign, scale, icpy, x, x3)
end module FieldQuant_class
```

The field quantity is updated at each step according to the type of boundary condition and solver. In our case, the field is updated using FFTs. We have implemented an HPF local subroutine to call an FFT solver from the machine library.

The Geometry class is used to instantiate different geometry objects. A sketch of the Geometry class module is given below.

```
module Geometry_class
  integer, private, parameter :: Ndim = 2
  type Geometry
    private
    real, dimension(2*Ndim) :: SpatRange
    real, dimension(Ndim) :: Meshsize
    integer, dimension(Ndim) :: Meshnum
  end type Geometry
```

```
contains
  pure logical function onbc_Geometry(this,ix,iy)
  pure logical function notin_Geometry(this,location)
  subroutine update_Geometry(this, inrange, msk)
end module Geometry_class
```

Here, SpatRange defines the range of the computational domain in each dimension. Meshsize and Meshnum give the numerical discretization of the domain.

Parallel loop implementation uses the HPF commands *Do Independent* and *Forall. Forall* is used in the case of 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.)

5 Application

The above model was 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 Cray T3E-900 using the Portland Group PGHPF compiler.

The Cray T3E-900 is a scalable, logically shared, physically distributed multiprocessor machine with a range of configurations up to thousands of processors [14]. Each node consists of a DEC Alpha 64-bit RISC microprocessor, local memory, system control chip and some network interfaces. The RISC microprocessor is cache-based, has pipelined functional units, and issues multiple instructions per cycle. The clock speed is 450 MHz. Each node has its own local DRAM memory with a capacity of from 64 Mbytes to 2 Gbytes. A shared, high-performance, globally addressable memory subsystem makes these memories accessible to every node. The nodes are connected by a high-bandwidth, low-latency bidirectional 3D torus system interconnect network.

The problem we tested here consists of 10 FODO periods. Our simulation used $250,000$ particles with a 128×128 field mesh. The field range is set by the range of particles in the accelerator. The performance for our code is given in Fig. 2. We measured the time spent on 5 major subroutines and scaled them with the number of processors. These are map1, map2, FFT, scattering and depositing subroutines. The speedup is calculated as the ratio of time measured on a given number of processors to the time measured on one processor. We see that the speedup increases to 32 processors and saturates beyond that. Checking the time spent on individual subroutines, we find that the grid-particle scatter subroutine reaches its maximum speedup at 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 at 32 processors. This is due to the global nature of the Fourier transform resulting in

communication overhead. The map1, map2 and depositing subroutine speedups keep on increasing till 64 processors. The map1 and map2 subroutines scale very well with increasing number of processors. However, the depositing subroutine does not scale well due to the communications in particle deposition. (Here, we have used a HPF library function *sum_scatter* in the depositing subroutine.)

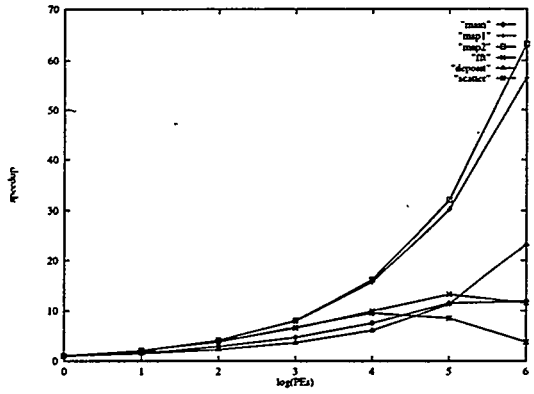


Fig. 2. The speedup as a function of number of PE on Cray T3E-900.

6 Conclusions

The application of an object-oriented design strategy with HPF should provide the benefits of better maintainability, reusability and extensibility. Using the concept of object, which is implemented using HPF modules, makes the code more resilient to requirement changes. For example, a new beam line element can be easily incorporated into the BeamLineElem module without affecting the other modules. Also, decomposing the beam dynamics accelerator system into physically meaningful modules helps to keep the code in good order while being tested and debugged. Moreover, individual class modules can also be reused in other applications such as plasma physics, fluid dynamics and electromagnetics.

Our design study was based on the current status of the PGHPF compiler technology. With further development of compilers, it is possible that the programming barriers we pointed out will disappear. In that event, our model can still be easily extended to adopt new features in the future.

To summarize, it appears to us that applying object-oriented design with HPF can achieve both good software quality and ease of parallel programming in scientific applications. However, we do not regard this as a competition with other object-oriented frameworks (e.g., using C++) but rather as a complement to these frameworks.

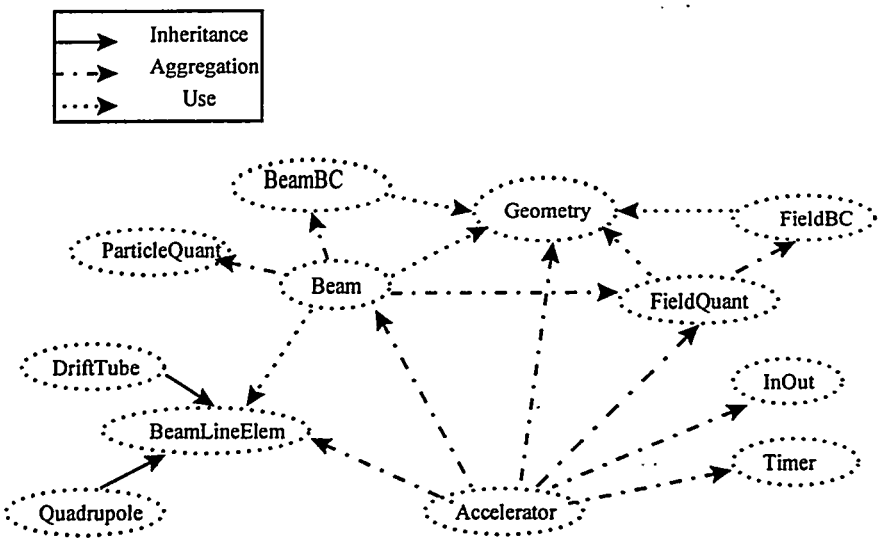
Acknowledgments

We acknowledge helpful discussions with Drs. Viktor Decyk, Graham Mark, and the POOMA team who also provided POOMA beam dynamics source code. This work was performed on the Cray T3E at NERSC and supported by the DOE Grand Challenge in Computational Accelerator Physics.

References

1. Wilson, G., Paul, L. (ed.): Parallel Programming Using C++, MIT Press, Cambridge (1996).
2. Williams, J. T., Reynders, J. V. W., Humphrey, W. F.: POOMA User Guide, http://www.acl.lanl.gov/pooma/doc/userguide, (1998).
3. Ryne, R., Habib, S., Parallel Beam Dynamics Calculations on High Performance Computers, In: Bisognano, J. J., Mondelli, A. A. (eds.): Computational Accelerator Physics, AIP Conference Proceedings 391, Woodbury, New York (1997) 377-389.
4. Ding, C. H. Q., HPF for Practical Scientific Algorithms, Preprint for Supercomputing '97, (1997).
5. Elisseev, V. V., Parallelization of Three-dimensional Spectral Laser-Plasma Interaction Code Using High Performance Fortran, Computers in Physics **12** (1998) 173-180.
6. Koelbel, C. H., Loveman, B. D., Schreiber, R. S., Steele, G. L., Zosel, M. E.: The High Performance Fortran Handbook, MIT press, Cambridge, (1994).
7. Ellis, T. M. R., Philips, I. R., Lahey, T. M.: Fortran 90 Programming, Addison-Wesley, Harlow, England (1994).
8. Gray, M. G., Roberts, R. M., Object-based Programming in Fortran90, Computers in Physics **11** (1997) 355-361.
9. Decyk, V. K., Norton, C. D., Szymanski, B. K.: Expressing Object-Oriented Concepts in Fortran 90, ACM Fortran Forum, Vol. 16, num 1, (1997).
10. Forest, E., Ruth, R. D., Fourth-Order Symplectic Integration, Physica D **43** (1990) 105-117.
11. Hockney, R. W., Eastwood, J. W., Computer Simulation Using Particles, Adam Hilger, New York, (1988).
12. Booch, G.: Object-Oriented Analysis and Design with Applications, Benjamin/Cummings, Menlo Park, CA, (1994).
13. PGHPF manual, http://www.nersc.gov/software/prgenv/compilers/pghpf/docs/, (1998).
14. http://www.cray.com/products/systems/cray3e/overview.html, (1998).

Fig. 1. The class diagram in beam dynamics study.


Inheritance
Aggregation
Use
BeamBC
Geometry
FieldBC
ParticleQuant
Beam
FieldQuant
DriftTube
InOut
BeamLineElem
Timer
Quadrupole
Accelerator