Charles D. Norton, Boleslaw K. Szymanski, and Viktor K. Decyk

Object-Oriented Parallel Computation for Plasma Simulation

Object-oriented techniques promise to improve the software design and programming process by providing an application-oriented view of programming while facilitating modification and reuse. Since the software design crisis is particularly acute in parallel computation, these techniques have stirred the interest of the scientific parallel computing community. Large-scale applications of ever-growing complexity, particularly in the physical sciences and engineering, require parallel processing for efficiency. Since its introduction in the 1970s, Fortran 77 has been the language of choice to model these problems, due to its efficiency, its numerical stability, and the body of existing Fortran codes. However, the introduction of object-oriented languages provides new alternatives for parallel software development. Fortran 90 adds modern extensions (including object-oriented concepts) to the established methods of Fortran 77. Alternatively, object-oriented methodologies can be explored through languages such as C++, Eiffel, Smalltalk, and many others. Our selection among these required a language that was widespread and supported across multiple platforms (particularly supercomputers) with strong compiler optimizations. C++, while not a "pure" object-oriented language, was our choice, since it meets these criteria.

Currently, the most promising technique for parallel programming combines a standard high-level language with an explicit message-passing library for interprocessor communication. However, languages can also be

extended with new constructs in direct support of parallelism. The principal explicitly parallel Fortran-based language is High Performance Fortran (HPF), which introduces directives for data placement and alignment. Additional research languages include Fortran D, Fortran 90D/HPF, Fortran M, Opus, and Vienna Fortran. Some of these languages support operations on virtual processors, which separate the problem partitioning and mapping from the physical processors. Research activities in object-oriented parallel languages include ACT++, C**, Charm++, Compositional C++, Concert, Concurrent Aggregates, Concurrent C++, COOL, DC++, DCE++, HPC++, Mentat, Parallel C++, pC++, POOL-T, and μC++. These languages support shared memory (address space is common to all processors), distributed memory (address space is local to each processor) and/or workstation cluster parallel environments. Each language adds extensions, typically to C++ and often with complex runtime systems, to support task and/or data parallel computation.

Many of the research-based modifications for parallelizing Fortran and C++ have very promising ideas, yet the proposed techniques may not receive overwhelming support unless clear, empirical, and measurable evidence establishes their benefits. Although valuable progress continues, until these methods become commonplace, as demonstrated by supercomputer manufacturer support and standards committees, most developers may remain apprehensive about adopting new languages. Thus, the future of scientific programming will depend on *establishing standards* and *recognizing educational trends* in software design. Even though Fortran 77 remains the most popular language in scientific computing, larger codes and generalization of computational kernels for reuse create an incentive to consider languages that support abstractions and modularity. Many of the new features of Fortran 90 can support object-oriented programming methodology. C++ has become an informal standard, as evidenced by widespread training programs in academia and industry. As a result, we believe that *standard Fortran 90 and C++ with standard message-passing libraries* provide an attractive basis for parallel programming.

We evaluate object-oriented programming methods in high-performance computing by discussing our software development experiences with plasma Particle in Cell (PIC) simulation skeleton codes. Beginning with the parallel Fortran 77 version, we convert the application into an object-oriented form using the Intel Paragon, IBM SP1/SP2, and Cray T3D distributed memory parallel computers. We also show how Fortran 90 supports object-oriented programming by mirroring every language feature used in the sequential C++ program. Our objective is to determine if the object-oriented paradigm is

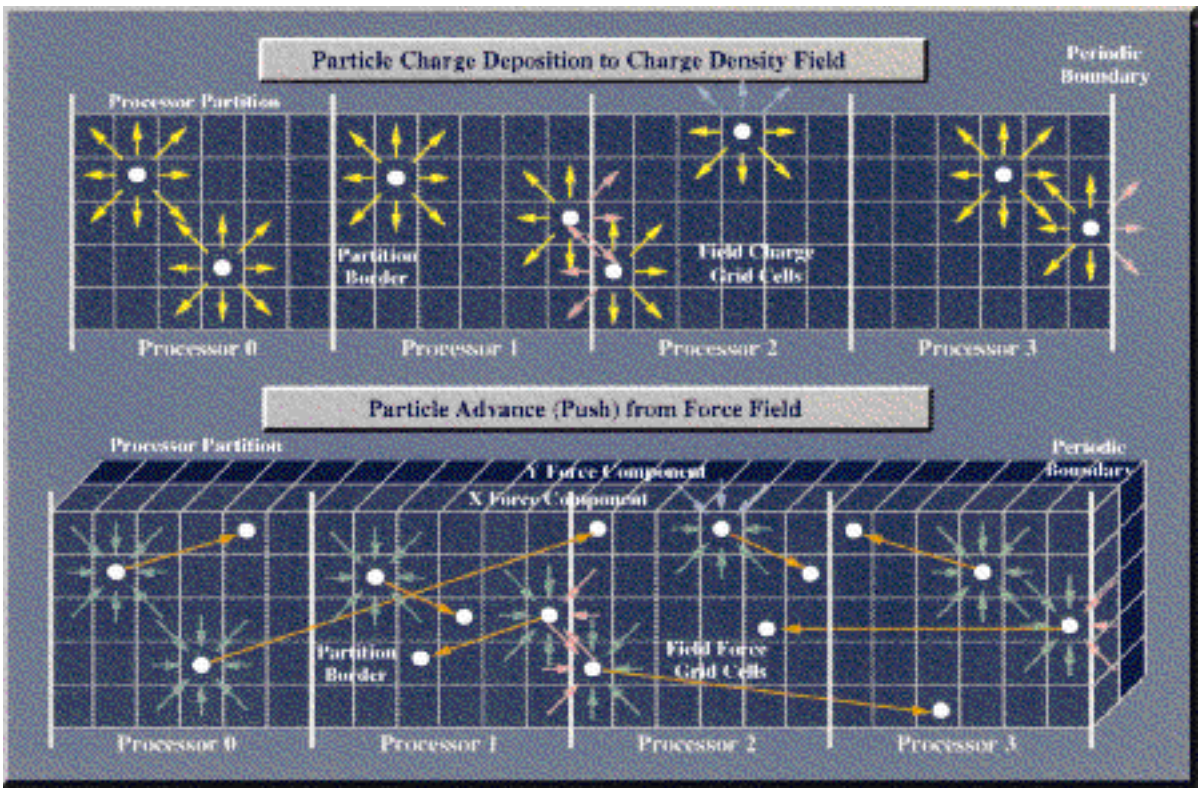


Figure 1. Particle/Field interaction in the plasma PIC algorithm (two-dimensional illustration). Red arrows indicate nonlocal charge/force data operations. Blue arrow operations are local due to the slab partitioning. Orange arrows show the new particle positions.

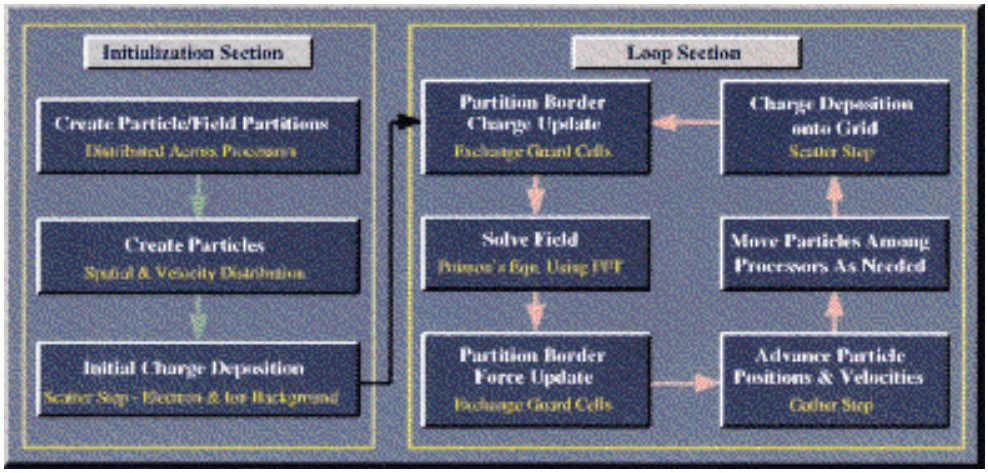


Figure 2. Plasma PIC computation loop overview: Diagnostic operations and extensions for load balancing are not shown

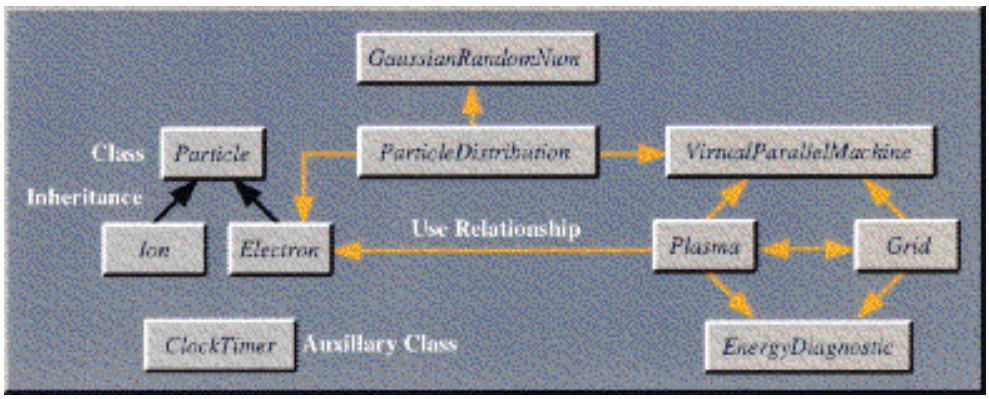


Figure 3. Object-oriented class hierarchy version 1: The classes utilize inheritance in the definition of specific particles and use-relationships to support interaction among abstractions

actually beneficial in high-performance scientific computation. Our study focuses on the practical issues encountered in software development on parallel machines, including programming abstractions, modifiability, portability (across message-passing libraries, machines, and compilers), numerical accuracy, and computational efficiency.

Overview of Plasma PIC Simulation

When a material is subjected to conditions under which the electrons are stripped from the atoms, acquiring free motion, the mixture of heavy positively charged ions and fast electrons forms an ionized gas called a plasma. Ionization can be introduced by extreme heat, pressure, or electrical discharges. Fusion energy is an important application area of plasma physics research, but more familiar examples of plasmas include the Aurora Borealis, neon signs, the ionosphere, and solar winds. The plasma Particle in Cell simulation model [1] integrates in time the trajectories of millions of charged particles in their self-consistent electromagnetic fields. The method assumes that particles interact with each other not directly, but through the fields they produce. Particles can be located anywhere in the spatial domain; however, the field quantities are calculated on a fixed grid. In our example application, only the electrostatic (coulomb) interactions are included.

The General Concurrent Particle in Cell (GCPIC) algorithm [8] partitions the particles and grid points among the processors of the multiple-instruction, multiple-data (MIMD) distributed-memory machine. The particles are evenly distributed among processors in the primary decomposition, which makes advancing particle positions and velocities in space efficient. A secondary decomposition partitions the simulation space evenly among processors, which makes solving the field equations on the grid efficient. As particles move among partitioned regions, they are passed to the processor responsible for the new region. For computational efficiency, field/grid data on the border of partitions are replicated on the neighboring processor to avoid frequent off-processor references. We illustrate the interaction between the particles and the field/grid in Figure 1 to show the data dependency that must be modeled in our class design. Particles scatter charge and gather force data to/from their nearest grid points. Force components from each dimension are required to advance particles to new positions.

We perform a Beam-Plasma instability experiment in which a weak low-density electron beam is injected into a stationary background plasma of high density, driving plasma waves to instability. Beam-Plasma interactions cause particle bunching, forming potential wells that are self-enhanced. This leads to particle trapping, creating vortices in phase space. The ions are modeled as a fixed neutralizing background. Although the number of particles per processor will vary during this simulation, the load remains sufficiently well balanced. This is not the case for all kinds of plasma simulations, where dynamic load balancing may be required [4]. (Our codes that support dynamic load balancing of the particles by rebuilding the

primary decomposition will not be discussed in this article.) An experiment such as this can be used to verify plasma theories and to study the time evolution of macroscopic quantities such as potential and velocity distributions. The GCPIC method can model a variety of sophisticated plasma simulations.

The Fortran 77 program is organized into two major sections, referred to as the *initialization section* and the *loop section*, as shown in Figure 2. The initialization section builds the particle and field partitions, constructs tables, and creates the initial particle distribution and charge density deposition. The loop section calculates the electric field forces using the Fast Fourier Transform (FFT) and Poisson's Equation, advances the particles under these forces, and finds the new charge density for the field at the grid points. Each loop represents a simulated time step during which diagnostics such as field, kinetic, and total energy are monitored.

Object-Oriented Simulation in C++ and Fortran 90

Various object-oriented designs have been proposed in plasma simulation [5, 10]. The following issues motivated our design:

- The impact of Fortran 77 program structure on class design.
- The interdependence between efficiency and class design.
- Numerical reliability compared to Fortran 77.
- The appropriate usage of C++ features and their expressibility in Fortran 90.

Plasma simulation inherently depends upon interactions between particles and fields. We seek to model this relationship from a physical and computational perspective with object-oriented methods. Although the Fortran 77 version is well organized, non-object oriented languages do not establish a relationship between the data and characteristic operations. We discuss how this relationship was captured in our C++ and Fortran 90 programs.

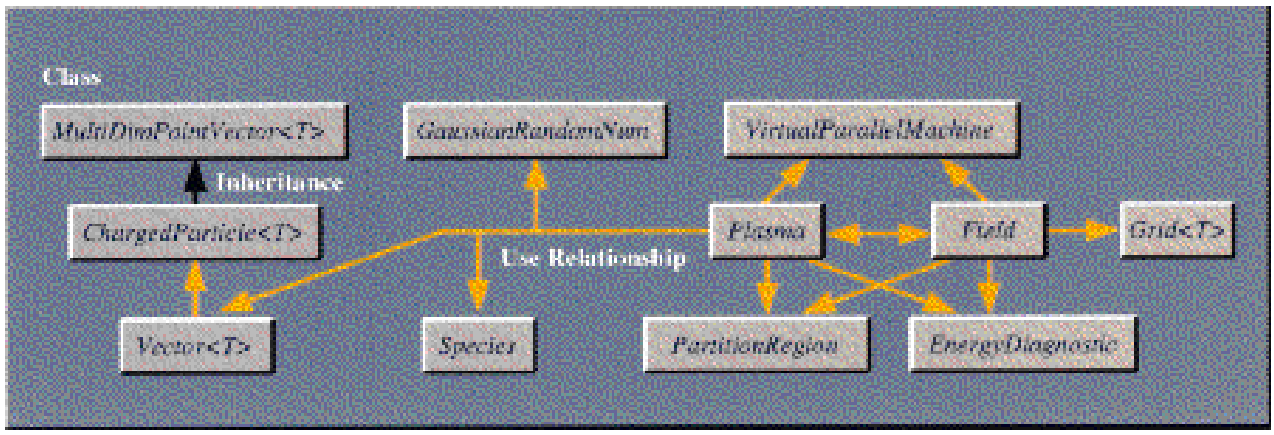
Analysis of the application and the Fortran 77 source identifies the field/grid, particles (individually and collectively), and diagnostics as potential modeling abstractions. However, organization of the classes requires consideration of their *interaction* (use-relationship) and *commonality* (inheritance).

Figure 3 shows the class hierarchy for the initial version of the one-dimensional C++ program. The `Particle` class, through its public methods (access functions), provides the interface for position and velocity information used by derived classes such as `Electron`. Inheritance implies that the `Electron` class has all the properties of the `Particle` class in addition to the specific features that define an electron. The `Plasma` class provides operations on the collection of electron objects that make up the plasma. Therefore, the `Plasma` class uses objects from the `Electron`, `Grid`, `EnergyDiagnostic`, and `VirtualParallelMachine` (VPMachine) classes. The `Grid` class provides operations to deposit charge and solve Poisson's Equation for the electric field, which requires an interaction with the `Plasma` class. The `EnergyDiagnostic` class is used by both the `Plasma` and `Grid` classes to record this diagnostic. Additionally, there is a need for classes that provide specialized services such as random numbers and timing measurements.

Although the original object-oriented version models the PIC simulation accurately, refinements and extensions can be introduced to ease the transition to higher dimensional codes. Furthermore, the original version makes salient assumptions regarding the simulation, such as fixed particle and field partitions, which may not hold in more general experiments that require dynamic load balancing. Hence, we designed an alternative class hierarchy in which the definition of a particle and the design of the simulation space have been reorganized. The new design reused much of the original code. We believe that such a refinement process is a necessary part of proper class hierarchy design for software generalization and modification.

Figure 4 shows the modified class hierarchy, which

Figure 4. Object-oriented class hierarchy version 2: This alternative model generalizes particles and fields by extending the original model with template classes



```
// Program Objects are Created
VPMachine vpm;
Vector < ChargedParticle > elec_pos( PTMAXNP ), elec_vel( PTMAXNP );
Species backgnd( N_BKELE_X, N_BKELE_Y, BKTHERMAL_VEL_X, BKTHERMAL_VEL_Y,
                 BKDRIFT_VEL_X, BKDRIFT_VEL_Y );
Species beam( N_BMELE_X, BMELE_Y, BMTHERMAL_VEL_X, BMTHERMAL_VEL_Y,
                 BMDRIFT_VEL_X, BMDRIFT_VEL_Y );
Plasma plasma;
EnergyDiag energy;
Field field( vpm, energy );
// Object Methods Partition the Plasma and Field, Distribute Particles and Deposit Charge
vpm.ParInit();
vpm.startclk();
plasma.Partition( vpm );
field.Partition( vpm );
plasma.UniformSpcMaxwellVelDist( elec_pos, elec_vel, backgnd, vpm );
plasma.UniformSpcMaxwellVelDist( elec_pos, elec_vel, beam, vpm );
field.ChargeDeposition( elec_pos, plasma, ChargedParticle::e_charge );
field.BackgroundIonDensity();
```

Figure 5. C++ initialization section sketch (two-dimensional program)

```
// Calculate Electric Field and Exchange Field Border Force/Charge
field.CalcEField( vpm, energy );
field.InitChargeDensity();
energy.ke( 0.0 );
// Push Particles and Update to New Partitions
plasma.Advance( elec_pos, elec_vel, field, energy, vpm );
plasma.UpdateDistribution( elec_pos, elec_vel, vpm );
// Deposit Charge and Ion Background with Energy Diagnostic
field.ChargeDeposition( elec_pos, plasma, ChargedParticle::e_charge );
field.BackgroundIonDensity();
energy.tote( energy.pe() + energy.ke() );
vpm.endclk( curOFile );
```

Figure 6. C++ loop section sketch: Method arguments illustrate use-relationships among classes

uses templates to operate on a vector space of particles. A particle is generalized by a vector that represents the position/velocity components in the corresponding dimensions. These vectors are inherited into a `ChargedParticle` class, which enhances the physical description of a particle. The plasma is modeled as a vector space of charged particles by the `Vector<ChargedParticle>` template class, which allows for vector operations on the collective group of particles. (Templates allow classes to be parameterized by an object type.) The `Species` class maintains specific information about the collective initial distribution conditions of particles, such as their thermal and drift velocities. The `EnergyDiagnostic` class collects and monitors plasma parameters associated with system energy. The `VPMachine` class aids in portability by parameterizing and encapsulating all of the machine-specific features.

The `Field` consists of computational grid points of `Grid<T>` template class objects, which unify force/charge data in multiple dimensions. A `PartitionRegion` object maintains field partitioning information across the processors. Operations associated with depositing charge and calculating the background ion density are members of the `Field` class, since they modify the field. The `Plasma` class performs collective operations on the vector space of particles. These include specifying spatial/velocity distributions, advancing particles under field forces, and redistributing particles when processor domain boundaries are crossed. Another `PartitionRegion` object specifies plasma partitions, since particles may be distributed differently from the field across processors. The initialization and loop sections of the program are shown in Figures 5 and 6, respectively. Note that the vector space of electrons is specified using two separate objects. Since mathematical vector operations on electron velocities (such as scalar multiplication) must not influence position components, this distinction is necessary.

The program classes directly represent physical and computational constructs through an organization that allows for interaction via use-relationships. Properly designed classes and objects allow straightforward modifications and extensions to the basic model. Additionally, the design of proper abstractions aids in the readability of the code. Readability in Fortran 77 can be difficult, since the underlying data is not bound to the associated routine and hence

large parameter lists are often required. The trade-off in using C++ involves the complex interdependence between class abstraction and its impact on efficiency; nevertheless, appropriate usage of language features can achieve satisfactory efficiency.

The new features of Fortran 90 provide support for the object-oriented methodology. An initial investigation modeled a curve-fitting application [3]. Our goal was to determine whether more extensive scientific computations could be represented. To address this issue, we have rewritten the initial sequential version of the one-dimensional C++ plasma PIC simulation code in Fortran 90 based on the C++ class hierarchy of Figure 3. We give examples of the Fortran 90 statements that can support object-oriented programming.

Figure 7 compares the C++ `Particle` class to an equivalent Fortran 90 module that allows for the encapsulation of data with associated operations. Access to the position and velocity data can be restricted to the functions defined as part of the class or module by using the `protected` and `private` qualifiers. Function overloading (which allows functions to share the same name but perform different operations based on the arguments) is modeled using the `optional` qualifier with the `present` statement in Fortran 90. Operator overloading is also supported.

Typically, the Fortran 90 `use` statement makes the public part of a module accessible to subprograms. However, inheritance can be supported through the use statement, which permits all or part of a module to be used in another module. C++ classes also have special member functions called constructors and destructors, which allow the automatic initialization and destruction of objects. Inheritance complicates the definition of constructors and destructors. Fortran 90 does not support the automatic initialization of module variables; however, this can be simulated by calling a user-defined `Create` routine once a variable is declared. Destructors would be more cumbersome to simulate through subroutine calls, since they would have to be provided in every context under which a module variable could be destroyed. Therefore, we did not model this concept in our Fortran 90 simulation. C++ static class variables can be modeled in Fortran 90 by using the `save` qualifier in a module; thus, only one copy of the module data will be used across multiple module instances. A snapshot of inheritance and static variables is shown in Figure 8.

The `friend` construct in C++ gives a class direct access to the private area of another class, usually for reasons of efficiency. Although Fortran 90 does not support friends, this property can be emulated using the `use only` statement, which selects specific parts of the module for usage. For instance, the particle advance routine is very time consuming, requiring field/grid information to update particle positions. By using C++ friends, `Grid` class data becomes directly accessible to the `Plasma` class advance routine, improving efficiency. The Fortran 90 `Plasma_m` module definition (not shown) uses the `use only`

```
class Particle {
    protected:
        float xpos, xvel;
    public:
        void pos_x( float pos ) { xpos=pos; }
        float pos_x() const { return xpos; }
};

// C++ Object Creation and Usage Example
Particle part;
part.pos_x(10.4);
val = part.pos_x();

! Fortran 90 Object Creation and Usage Example
type (particle) part
void = pos_x(part. 10.4)
val = pos_x(part)
```

```
module Particle_m
    type particle
      private
      real xpos, xvel
    end type particle
    contains
    real  function pos_x(part,pos)
    type (particle) part
    real, optional::pos
    if (present(pos)) then
        part%xpos = pos
        pos_x = 0.
    else
        pos_x = part%xpos
    endif
    end function pos_x
end module Particle_m
```

Figure 7. C++ particle class and Fortran 90 particle module sketch: Data can be encapsulated in Fortran 90 by using the `derived type` within a module. The `module` acts as a class, providing an interface to member data through routines defined within the `contains` statement

```
class Electron : public Particle {
    public:
        static const float charge;
};
```

```
module Electron_m
        use Particle_m
        real,  parameter :: CHARGE = −1.0
        save
end module Electron_m
```

Figure 8. Sketch of inheritance and static data usage in C++ and Fortran 90

```
class Grid {
    protected:
        float *g, *fx;
    public:
        friend class Plasma;
        Grid() { q = new float[SYSLEN_X];
                 fx = new float[SYSLEN_X];
        }
        void AddIonDensity() {
        for (register int i=0; i<SYSLEN_X; i++}
                 q[ i ] += Ion::qion;
        }
};
```

```
module Grid_m
    use Ion_m
    real, dimension(:), allocatable::q
    real, dimension(:), allocatable::fx
    save
    contains
    subroutine  Grid_Create()
        allocate (q(NX),stat=ierr)
        if (ierr.ne.0) void=FreeStoreException()
        q = 0.
        allocate (fx(NX),stat=ierr)
        if (ierr.ne.0) void=FreeStoreException()
        fx = 0.
    end subroutine  Grid_Create
    real function  Grid_AddIonDensity()
        q = q + qion
        Grid_AddIonDensity = 0.
    end function  Grid_AddIonDensity
end module Grid_m
```

```
main()
{
Plasma plasma; Electron elec[NP];
Grid grid; EnergyDiag energy;

plasma.Advance(elec, grid, energy, NP);
grid.depositCharge(elec, Electron::charge, NP);
}
```

```
program beps1k
use Electron_m, EnergyDiag_m, Plasma_m
use Grid_m, only: Grid_Create, Grid_Setup,
1Grid_InitChargeDensity, Grid_AddIonDensity,
2Grid_CalcEField, Grid_DepositCharge

type (particle) elec(NP)
type (energy) energ
call Plasma_Advance(elec,energ,NP)
call Grid_DepositCharge(elec,CHARGE,NP)
```

statement on the `Grid_m` module field data, simulating C++ friends. However, in contexts where the `Grid_m` module field data should not be accessible, any other module routines of interest must be used explicitly via the `use only` statement. Modeling the friend construct of C++ in this way was the only awkward construct encountered.

Dynamic memory allocation allows for flexibility in data structure design and manipulation. In C++, the `new` statement is used to dynamically allocate memory. In Fortran 90 we can declare a variable to be `allocatable` where the `allocate` statement in a module subroutine provides physical memory. Array operations in Fortran 90 allow mathematical operations on entire arrays. In C++, the associated operators must be overloaded explicitly with boundary checking, since no effort is made to guard against illegal indexing of arrays. An example is shown in Figure 9.

The methods for performing operations on C++ objects and on Fortran 90 module variables are related. In C++, the member functions are bound to the object using the syntax *object*.`MemberFunction()`. In Fortran 90, variables are created from modules (within the scope of the `use` statement) using the `type` statement; hence they are not bound to module functions and subroutines. The module variable must be provided as an argument to its functions and subroutines using the syntax **call** `MemberFunction`*(variable, ...)*. This resembles the manner in which the C++ method calls are actually translated by most compilers. Additionally, Fortran 90 performs type checking on function arguments so the proper variable type is applied to a valid associated module member. A small illustration of this is shown in Figure 10 for the C++ and Fortran 90 programs.

Figure 9. Dynamic memory allocation: A portion of the `Grid` class is shown with the corresponding `Grid` module illustrating dynamic memory allocation. Array operations are also illustrated in the `Grid_AddIonDensity` routine

Figure 10. Illustration of C++ object and Fortran 90 variable creation and usage

It should be noted that certain important features of C++ have not been used. In particular, we did not use `virtual functions`, which allow the runtime selection of the routine that will be called on an object. Fortran 90 can support virtual functions by the `generic subprogram` feature. When an argument is pro-

vided to a generic subprogram, the appropriate routine is executed based on the argument's type.

Program Development Experiences Across Compilers and Machines

Our development environment consists of the Intel Paragon XP/S, IBM SP1/SP2, and Cray T3D distributed memory MIMD parallel machines. Each Paragon node contains two or more i860 computational processors and a message-passing processor. Interprocessor communication over the rectangular mesh uses the NX message-passing library. The SP series uses RS6000 processors interconnected via a high-performance switch (as well as Ethernet) with the MPL communication library. The T3D supports shared and distributed memory paradigms using DEC Alpha processors over a three-dimensional toroidal-wrap topology. Communication on the T3D uses a modified version of PVM. The Paragon and SP series (and soon T3D) also support the Message Passing Interface (MPI) standard. We used GNU g++ and Intel C++ on the Paragon, IBM xlC on the SP1/SP2, and Cray C++ on the T3D.

The Fortran 77 versions of the plasma simulations compiled without difficulty across these machines, due to the extensive support provided for this language in scientific computing. A major goal of our C++ development effort was to maintain *machine-* and *compiler-independent* versions of the programs. Modifications to system files were introduced to support g++ on the Paragon; also, template usage required special attention in code generation across compilers.

The non-template based one-dimensional PIC program performed properly under v2.4.5 of the GNU g++ compiler on the Paragon, but when recompiled using v2.5.7, incorrect energy diagnostics were reported. Although porting the two-dimensional template-based program from the SP1 to the Paragon was straightforward, numerical errors arose in the template references on the Paragon, which disappeared in v2.6.1 of g++. These compiler inconsistencies resulted in five months of lost development time. The Intel C++ compiler performed well in our two-dimensional and three-dimensional template-based programs.

The IBM SP1/SP2 and xlC C++ compiler performed extremely well; however, the SP1 would hang indefinitely, failing to release the processors, after large simulations executed to completion. Although this issue could not be experimentally characterized, IBM representatives stated that recent system software releases have resolved this problem. In fact, this issue did not occur on the SP2. Template instantiation and usage were never a problem with the xlC compiler.

The Cray T3D C++ 1.0 compiler could not instantiate template classes used across multiple files. Interestingly enough, the identical program did compile correctly on the Cray Y-MP. Cray responded to our difficulty and installed Cray C++ 1.0.3.1 in December 1994. The template class instantiation problem was corrected, yet problems with the creation of template functions still persisted. We removed the template functions from the source program to force compilation, but the executable would not run on the T3D. The identical program works correctly on the Paragon and the SP series. Our difficulties with the C++ compiler on the T3D remain unresolved as of June 1995. Software problem reports have been filed and are under investigation.

Experiences in Portability

The `VPMachine` class provides a standard interface to the machine-specific message-passing environment and system calls. Utility routines, such as timing and processor communication routing operations, are also provided with facilities to allow object-based interprocessor communication. Thus, rather than performing a send/receive on an array of floating-point numbers representing particle positions, we actually transmit full `Particle` objects. This preserves the object-oriented nature of the simulation environment. As MPI becomes more widespread, we expect machine-specific classes will decrease in importance; yet the ability to perform message-passing on objects should remain valuable. We maintain MPI versions of our programs, as well as an MPI virtual machine class.

Program design and testing evolved simultaneously across multiple compilers and machines using the `VPMachine` class; hence, our codes were easily ported among machines. This was particularly useful in finding and reporting bugs in the GNU and Cray compilers. Without this capability, a C++ code developed on one machine with a single compiler would have required organizational changes for portability.

Experiences with Efficiency

Fortran is well known for its efficiency, while C++ has a reputation (perhaps unjustified) for being much less efficient. Designing efficient and portable C++ codes is difficult due to differences in compiler implementations. Inlining is touted as "the solution" to the overhead associated with calling methods on objects. Programmers must note that compilers are free to ignore the `inline` directive. One major source of inefficiency results from the casual use of the mathematical operations. Our initial sequential C++ plasma simulations executed five times slower than the sequential Fortran 77 versions due to inefficiencies in the standard C++ `pow()` routine. We realized that Fortran could optimize this routine based on the arguments to the function, so we overloaded the `pow()` routine in C++ to include this distinction. This change reduced the total time used for exponentiation from 65% of the total computation time to less than 1% for the sequential C++ programs.

Memory overhead and data access time also contribute to inefficiency. Many plasma simulation models represent particles by dynamic lists, which severely restricts the size of the simulations due to the memory consumed by pointers. Our particle representations use object arrays, which require special algorithms to maintain data structure consistency

Table 1. Paragon XP/S, SP2 and T3D Basic System Characteristics (From Specification reports).

Architecture Features	Intel Paragon	IBM SP2	Cray T3D
Processor Power	50 MHz	66.7 MHz	150 MHz
Network Speed	175 MB/sec	40/80 MB/sec (switch)	300 MB/sec

when particles cross processor partitions. This approach allows for larger simulations, since arrays use memory more productively than lists. Static class variables also optimize memory, since data, such as the electron charge, is not replicated over millions of electron objects; only a single copy is stored.

Active object creation or usage results in an overhead that is larger for inheritance relationships in class design than in use-relationships. Consequently, whenever possible, we defined use-relationships in such a way as to increase the efficiency of interaction among class abstractions. Finally, when writing numerical routines in an object-oriented framework, mathematical functions should be designed to work *within* an object class structure; they do not need to be object-oriented themselves. The FFTs and Poisson solver do not belong to mathematical classes; however, they do operate on simulation class objects.

The Fortran programs have been tuned for efficiency in ways that can be awkward for C++ programs. For example, in Fortran, arrays can be used directly in message-passing parameters, eliminating the need for temporary buffers and data copying involved in user send/receive calls. The C++ versions do not make data directly accessible to communication routines, often due to template-related issues; hence buffers are required. These buffers collect the transmitted data, which are then assigned to the associated object using its interface, to preserve encapsulation and protect nonpublic data. Although direct access to protected data by the message-passing routines would violate encapsulation, this may be appropriate for efficiency reasons, as with usage of the C++ friend statement. However, our field model consists of grid template points that maintain both the charge and multidimensional force data. The interprocessor data-flow requirements in the GCPIC algorithm require transmission of charge data and force data as separate operations.

Transmission of charge (force) data directly to the template field will overwrite the force (charge) data, since the memory for each grid template point is allocated contiguously. The `derived datatype` feature of MPI, which allows transmission of noncontiguous data, can address this issue. Nevertheless, this illustrates how the program abstraction features of C++ can influence efficiency in accessing data.

Reliability Issues

Many useful features for programming abstraction are provided by C++; nevertheless, the reliability of existing compilers must be considered. Reliability issues are noticed most clearly during the compilation process. Valid C++ programs that compiled correctly under one compiler could not be moved verbatim to other compilers. Difficulties with memory alignment and problems with linkers not resolving every external constant reference also arose. These issues cannot be detected at compile-time, requiring extensive run-time analysis followed by minor alternative implementation techniques.

In general, C++ can be stable, but as more sophisticated programming techniques are used, compiler bugs can severely restrict development. Often program development on the parallel machines was delayed while compiler issues were being resolved. In such circumstances, the ability to continue development using simulators or sequential machines is of great importance.

Comparisons Among Programming Paradigms

Developing the plasma PIC simulation in Fortran 77, C++, and Fortran 90 allows comparisons among the paradigms. Although Fortran 77 remains robust across compilers and machines, increasingly extensive work in simulation continues to strain the capabilities of this language. Grand Challenge-type problems require new approaches and methodologies, which must be supported by the implementation language. Representing abstractions is a prominent issue causing object-oriented methods to gain acceptance as a viable alternative for high-performance parallel computation. An unresolved question is whether it is always possible to decompose a problem into appropriate classes with communicating objects that interact. We argue that parallel computation is fundamentally dependent upon interactions and programming abstractions and that C++ and Fortran 90 can support these viewpoints very well. C++ is a young, evolving language that requires more extensive support by compiler developers and machine manufacturers before its full potential in scientific programming can be realized. Fortran 90 provides the robustness of Fortran 77 with programming abstractions relevant to the object-oriented methodology. This is an exciting language, and our early experience indicates it shows a lot of promise.

The parallelization strategy in our application, partitioning data across processors with message passing for communication, is the same across Fortran 77 and C++ paradigms. Development and implementation are where object-oriented methods are beneficial, since abstractions relevant to the application can be created to simplify the programming process. The development of these abstractions does represent a major portion of the effort involved in rewriting an existing Fortran 77 program into an object-oriented frame-

work. Construction of the C++ programs required an in-depth understanding of the design of the Fortran 77 code and the characteristics of the physical application. With this understanding, the C++ versions were written from scratch. In contrast, the object-oriented features of Fortran 90 were incrementally introduced into the existing sequential Fortran 77 program (using the class hierarchy of the sequential C++ program). As a result, construction of the Fortran 90 program was achieved in only a few days.

Conceptual abstractions introduced by object-oriented methods can be extended into benefits toward the programming of parallel distributed memory machines. Maintaining distributed data requires mechanisms for preserving consistency across processor boundaries. Using object-oriented paradigms, the *definition* of classes that represent distributed data, such as the field and particle classes, can provide features to maintain consistency. Abstractions, such as the `VPMachine` class, support parallel programming with object methods that transport data using its full object type. Fortran 77 or C implementation paradigms with message-passing calls differ from the object paradigm due to abstraction modeling. Implementation of the abstractions at the lowest level must be created to work within the class hierarchy and features of the architecture, but once they are created, many parallel programming details can be information-hidden.

The efficiency of Fortran programs is commonly cited as the major benefit over C++, yet this was not as important an issue as compiler stability across machines. Abstraction representation in C++ class hierarchies must allow for ease of extension. Unfortunately, hierarchies are nearly impossible to design correctly on the first attempt. Moreover, when the design is poorly organized, it is difficult to modify it without triggering something close to a complete redesign. Reuse of the relevant portions of the early design is often possible, but if the new class hierarchy cannot be defined with clean interfaces, the best approach is to redesign it from the beginning.

The C++ program syntax necessarily caused our programs to be longer (about 2.5 times) than the equivalent Fortran 77 versions in the plasma simulations. To design efficient C++ programs, the programmer must be aware of many "behind the scenes" operations that take place during execution. The learning curve for C++ is much longer than that for Fortran 90. Readability, while dependent on the style, taste, and experience of the programmer, can be enhanced through the object-oriented methodology. Viewing programs in terms of object types with well-defined operations, such as particles, fields, and partitions, adds clarity as code is shared and reused. C++ programs, in general, are necessarily slower than Fortran 77, since optimization is more limited across pointer structures than across static arrays. This issue must be weighed against the costs of program maintenance, which is a growing concern of many application programmers.

Our experience indicates that the efficiency of Fortran 77 and the abstraction modeling capabilities of C++ are desirable features for scientific programming. The new constructs of Fortran 90 modernize Fortran 77 with features to represent abstraction.

Parallel Simulation Results and Performance

In our Beam-Plasma instability experiment, we measure the field, kinetic, and total energies of the system at each simulated time step. Since the original Fortran 77 codes have been well benchmarked [2], we will restrict our performance overview to rather arbitrarily selected cases across the machines of interest. These results are intended only to illustrate how this code performs in Fortran 77 and C++ with standard optimization (-O) on various machines using the same number of processors. Although these architectures differ in technical specifications, we show two basic parameters, the processor power and the interconnection speed, in Table 1. In Table 2 we show processor simulation results for a few million particles across various simulation dimension sizes.

Additional simulation comparisons are shown in

Table 2. Paragon XP/S, SP1/SP2 Multimillion Particle Parallel Performance Characteristics.

Machine	Number of Processors	Language and MP Library	Number of Particles	Time (seconds)
Intel Paragon XP/S	32	Fortran 77 (NX)	4,505,600 (1D)	231.00
Intel Paragon XP/S	32	C++ (NX)	4,505,600 (1D)	377.00
IBM SP1	16	Fortran 77 (MPL)	3,571,712 (2D)	802.00
IBM SP1	16	C++ (MPL)	3,571,712 (2D)	1228.00
IBM SP2	16	Fortran 77 (MPL)	3,571,712 (2D)	364.00
IBM SP2	16	C++ (MPL)	3,571,712 (2D)	715.00
IBM SP2	32	Fortran 77 (MPL)	7,962,624 (3D)	1649.00
IBM SP2	32	C++ (MPI)	7,962,624 (3D)	2797.00
Cray T3D	32	Fortran 77 (PVM)	7,962,624 (3D)	2582.50
Cray T3D	256	Fortran 77 (PVM)	127,401,984 (3D)	5637.10

Table 3. Paragon XP/S and SP1/SP2 Fixed Problem Size Parallel Performance Characteristics.

	Execution Time (seconds)					
Processors	**Paragon 1D**		**Paragon 2D**		**Paragon 3D**	
	Fortran 77	C++	Fortran 77	C++	Fortran 77	C++ (MPI)
4	115.52	269.25	392.17	998.19	1,542.64	5,681.90
8	66.57	140.27	201.57	498.38	767.22	2,882.84
16	42.06	76.55	112.47	259.87	393.41	1,483.62
32	33.11	47.12	70.09	141.15	N/A	N/A
	450,560 Particles 2,048 Grid Points		327,680 Particles 8,192 Grid Points		294,912 Particles 32,758 Grid Points	
	Execution Time (seconds)					
Processors	**SP1 2D**		**SP2 2D**		**SP2 3D**	
	Fortran 77	C++	Fortran 77	C++	Fortran 77	C++ (MPI)
4	175.95	412.00	119.34	257.00	392.25	826.00
8	92.82	205.00	71.49	133.00	164.11	400.00
16	53.39	111.00	46.55	80.00	87.05	192.00
	327,680 Particles 8,192 Grid Points				294,912 Particles 32,758 Grid Points	

Table 3. Note that the Paragon 3D C++ (MPI) timings are much larger than the Fortran 77 timings. These runs were performed with the Intel C++ compiler, which seemed to "ignore" our more efficient overloaded mathematical routines. The remaining Paragon runs used GNU g++ v2.6.1. Additionally, we did not make any attempts to manipulate cache usage in the C++ programs. Results of work performed in this area for sequential PIC codes indicate up to 90% of Fortran 77 efficiency [11].

The C++ version appears more competitive as more processors are used, since the problem size remains fixed, as illustrated in Figure 11. This shows how performance results can be misleading, since the ratio of computation to communication dropped with decreasing numbers of particles within critical loop iterations. Outstanding C++ compiler problems prevented us from providing simulation results for the Cray T3D. IBM xlf90 was used for the sequential Fortran 90 program, but the lack of compilers for our parallel machines prevented timing comparisons to the Fortran 77 and C++ codes. We can give some indication of the performance of the object-oriented sequential Fortran 90 code by comparing it to that of the Fortran 77 and C++ one-dimensional sequential codes, as shown in Table 4. (The original sequential C++ program executed correctly with GNU g++ v2.6.3 on the Sun SPARCstations. When recompiled on the RS6000 under g++ v2.5.8 and IBM xlC, incorrect numerical results in complex arithmetic were detected. Reorganizing the memory layout of the data structures corrected this problem). Modeling the C++ technique of invoking a method to access private data probably contributed a performance overhead to the Fortran 90 program.

Table 4. One-Dimensional Sequential Performance Characteristics.

Machine	Language	Compiler	Number of Particles	Time (seconds)
IBM RS6000	Fortran 77	IBM xlf	450,560	245.49
IBM RS6000	Fortran 90	IBM xlf90	450,560	364.25
IBM RS6000	C++	IBM xlC	450,560	508.00

Conclusion and Commentary

We have given an overview of the design of C++ and Fortran 90 skeleton particle simulation codes based on existing Fortran 77 codes and discussed the design concepts involved in reorganizing the Fortran program into an object-oriented form. Additionally, we have given performance results that indicate that the execution speed of C++ may be acceptable, given the organizational advantages. The codes were designed with both

execution and implementation scalability in mind.

When considering design comparisons between Fortran 77 and C++, we noticed that the class structure provides a programming perspective that reflects the problem domain. Modifications and extensions to the object-oriented version are straightforward; however, classes must be carefully designed with extensions in mind. The object-modeling paradigm also enhances code readability through well-defined interfaces enforced by the C++ syntax.

There is a growing interest in the development of C++ class libraries for parallel simulation of plasma and other applications. The OOPS C++ class library [10] defines four main groups of objects for the parallel architecture, field, particles, and I/O. High-level objects hide details of the specific machine in use from the user, providing an interface that looks data parallel but which is actually message-passing based. C++-based libraries have also been developed for VLSI CAD applications [9], finite-element/finite-volume computations (DIME++) [12], and materials science (LPARX) [7]. Generally, library-based approaches try to preserve existing C++ codes rather than introducing new languages or language extensions.

We were amazed by the Fortran 90 statements that reflect object-oriented statements in C++. The most general features, such as inheritance and encapsulation, are covered, as well as certain details, including static member variables and class friends. We illustrated how various Fortran 90 statements compare to the equivalent C++ statements in the sequential PIC code. In most cases, the mapping was straightforward and precise, although the implementation of class friends was somewhat awkward. Given these new language statements, we believe that Fortran 90 (and HPF) programmers would benefit from a knowledge of object-oriented methods when putting these constructs into practice.

Our experience indicates that the object-oriented programming paradigm is beneficial in scientific computation when class hierarchy decisions are made with care. The two-dimensional codes were designed with modest extensions from the one-dimensional versions. Our three-dimensional C++ codes were quickly developed from the two-dimensional template versions. This demonstrates the usefulness of object-oriented programming methods in high-performance computing. Simulation extensions were based on modifying existing abstractions supported in a parallel environment. Since these abstractions incorporated the scientists' view of simulation, we experienced a rapid increase in the programming and reliability of new codes in higher dimensions. Although we described our experiences with object-oriented methods for an experiment that does not require load balancing, we are exploring ways in which this methodology can enhance the programming of more dynamic problems. Our ongoing research on using C++ for runtime efficiency of unstructured and irregular parallel computations, as in more advanced plasma simulations, is the focus of our current and future efforts.

The C++ programming language is still evolving; greater conformance to standards and numerical computational kernels are needed. Programming in an object-oriented manner takes practice and patience. As numerical classes are introduced and as new techniques are found to improve the efficiency of C++ programs, the benefits of object-oriented design will influence scalable high-performance computing. In assessing the trade-offs between Fortran efficiency and object-oriented design, the increasing costs of software

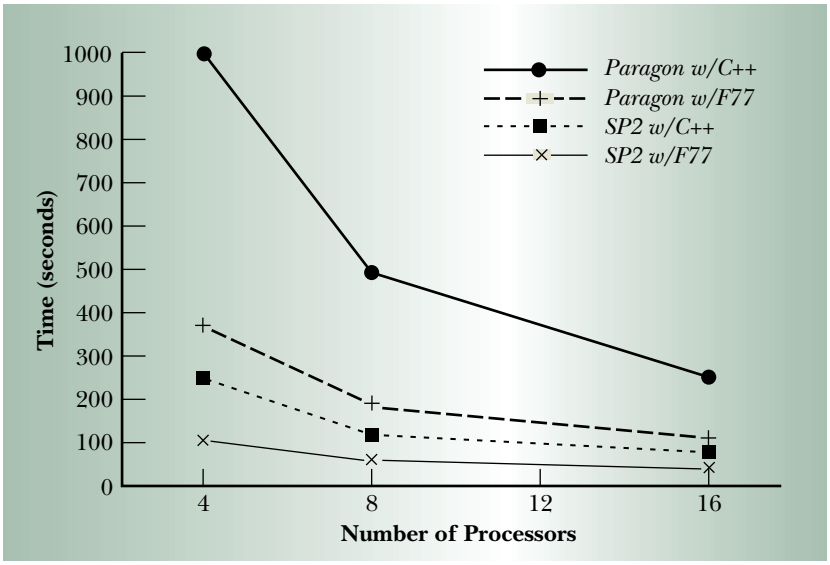


Figure 11. Paragon and SP2 two-dimensional Fortran 77 and C++ execution profiles for a fixed problem size

maintenance must be considered. The ability to reuse existing software and to develop computation kernels represents a growing need as high-performance computing becomes more complex. Object-oriented methods can help to achieve this result.

Acknowledgments
We appreciate the technical assistance of Edith Huang, Nooshin Meshkaty, and Jack Miller from JPL and Mark Miller from RPI regarding the compilers on the parallel machines. We also thank Joyce Brock and Eva Ma from RPI for their valuable insights during the preparation of this article.

References

1. Birdsall, C.K. and Langdon, A.B. *Plasma Physics via Computer Simulation*. Hilger Series on Plasma Physics, Hilger, New York, 1991.
2. Decyk, V.K. Skeleton PIC Codes for Parallel Computers. *Computer Physics Commun. 87*, 1/2 (May 1995), 87–94,
3. Dupee, B.J. Object Oriented Methods using Fortran 90. SIGPLAN Fortran Forum (Mar. 1994), 21–30.
4. Ferraro, R.D., Liewer, P.C., and Decyk, V.K. Dynamic load balancing for a 2D concurrent plasma PIC code. *J. Comput. Physics* 109, 2 (Dec. 1993), 329–340.
5. Haney, S.W. and Crotinger, J.A. C++ proves useful in writing a Tokamak systems code. *J. Computers in Physics 6,* 5 (Sept./Oct. 1991), 450–455.
6. High Performance Fortran Forum. *High performance Fortran language specification,* version 1.0 ed., May 1993. Tech. Rep. CRPC-TR92225, Rice University, Houston, January 1993.
7. Kohn, S.R. and Baden, S.B. The parallelization of an adaptive multigrid Eigenvalue solver with LPARX. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing* (San Francisco, Feb. 15–17, 1995), pp. 552–557.
8. Liewer, P.C., and Decyk, V.K. A general concurrent algorithm for plasma particle-in-cell simulation codes. *J. Computational Physics, 85* (1989), 302–322.
9. Parkes, S., Chandy, J.A., and Banerjee, P. A library-based approach to portable, parallel, object-oriented programming: Interface, implementation and application. In *Proceeding of Supercomputing '94 (*Washington, D.C., Nov. 14–18, 1994) pp., 69–78.
10. Reynders, J.V.W. Object-oriented particle simulation on parallel computers. In *Proceedings of the Fifteenth International Conference on the Numerical Simulation of Plasmas (*King of Prussia, Penn., 1994), pp. 1–4.
11. Turner, M. Experience with PIC-MCC and C++. In *Proceedings of the Fifteenth International Conference on the Numerical Simulation of Plasmas (*King of Prussia, Penn., 1994).
12. Williams, R.D. *DIME++: A parallel language for indirect addressing.* Tech. Rep. CCSF-29, CCSF, California Institute of Technology, Pasadena, CA, January 1993.

About the Authors:
CHARLES D. NORTON is a Ph.D candidate in the Department of Computer Science at the Rensselaer Polytechnic Institute. Current research interests in parallel computation include scientific computing, algorithms, object-oriented methodology and languages, and visual programming environments. email: nortonc@cs.rpi.edu; http://www.cs.rpi.edu/~nortonc

BOLESLAW K. SZYMANSKI is a professor of computer science and a founding member of the Scientific Computation Research Center (SCOREC) at Rensselaer Polytechnic Institute. Current research interests include the design and optimization of compilers and algorithms for parallel and distributed processing and the simulation and modeling of ecological systems. email: szymansk@cs.rpi.edu; http://www.cs.rpi.edu/~szymansk

Authors' Present Address: Amos Eaton Hall, Rensselaer Polytechnic Institute, 110 8th Street, Troy, NY 12180-3590.

VIKTOR K. DECYK is a computational physicist at the UCLA as well as a member of the technical staff at the Jet Propulsion Laboratory. Current research interests include computational plasma physics, particularly the use of particle simulation to model fusion plasmas and basic plasma processes.

Author's Present Address: Physics Dept., UCLA, Los Angeles, CA 90024-1547. email: decyk@physics.ucla.edu

This work is supported by the National Aeronautics and Space Administration under Grant NASA NGT-70334. The content does not necessarily reflect the position or policy of the U.S. Government. No official endorsements should be inferred or implied. Access to the Intel Paragon and Cray T3D at the Jet Propulsion Laboratory was provided by NASA's Offices of Aeronautics, Mission to Planet Earth, and Space Science. The IBM SP1/SP2 was provided by the Scientific Computation Research Center (SCOREC) at Rensselaer.