OBJECT-ORIENTED PROGRAMMING PARADIGMS IN SCIENTIFIC COMPUTING

By
Charles D. Norton

An Abstract of a Thesis Submitted to the Graduate Faculty of
Rensselaer Polytechnic Institute
in Partial Fulfillment of the Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

The original of the complete thesis is on file in the
Rensselaer Polytechnic Institute Library

Approved by the
Examining Committee:

_____
Dr. Boleslaw K. Szymanski, Thesis Adviser

_____
Dr. Viktor K. Decyk (Physics/UCLA), Member

_____
Dr. Joseph E. Flaherty, Member

_____
Dr. Franklin T. Luk, Member

_____
Dr. David R. Musser, Member

Rensselaer Polytechnic Institute
Troy, New York

August 1996
(For Graduation December 1996)

OBJECT-ORIENTED PROGRAMMING PARADIGMS IN SCIENTIFIC COMPUTING

By
Charles D. Norton

A Thesis Submitted to the Graduate Faculty of
Rensselaer Polytechnic Institute
in Partial Fulfillment of the Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

Approved by the
Examining Committee:

Dr. Boleslaw K. Szymanski, Thesis Adviser

Dr. Viktor K. Decyk (Physics/UCLA), Member

Dr. Joseph E. Flaherty, Member

Dr. Franklin T. Luk, Member

Dr. David R. Musser, Member

Rensselaer Polytechnic Institute
Troy, New York

August 1996
(For Graduation December 1996)

Contents

List of Tables

List of Figures

Preface

There is little argument that the programming of scientific applications, particularly on distributed memory parallel machines, remains difficult. Scientists are still productive, but this productivity comes at the high cost of designing and developing applications that are hard to maintain, modify, share with collaborators, explain, and scale to larger problems. Many of these challenges stem from applying languages and techniques that do not promote abstraction in programming. The human cognitive ability to unify and generalize detailed concepts into abstract representations is the most powerful tool available in scientific and engineering disciplines. Unfortunately, we have not yet been successful in transferring this technique into a general application development process. Programming via abstraction is not common practice.

One of the most promising areas of opportunity to address abstraction in scientific programming involves object-oriented technology. The advanced techniques of this approach can bring coherency to the design and programming process. Issues of interest include application-oriented design of scientific programs and language selection within an environment where new machines and languages are introduced and are rapidly changing. This represents a serious concern for software development, vital to the immediate and long-term success of production-oriented application design. These issues are even more critical as individuals consider how best to prepare themselves for the use of modern programming methodologies in the hope of being more productive.

Many scientific programmers are contemplating issues of language selection to in-

crease productivity, communication and understanding. Since decision making could not be based on an empirical comparative analysis of a single application written using a variety of techniques and modern languages, other factors influenced these decisions. This may have included our current experience, educational trends, management concerns, or even interest in the latest and fastest growing technologies—high performance scientific JAVA has even been proposed.

Our research, for the first time, allows direct comparisons among the design, development and performance of an application based on existing and emerging programming methodologies. As a result, interest in this work has already generated attention, both within the United States and internationally. Recent requests for information have come from Brazil, Japan (Electrotechnical Laboratory at Tsukuba, Real World Computing Partnership, and University of Tokyo), Switzerland (Ecole Polytehnique Fédérale de Lausanne) and France (Institut National de Physique Nucléaire et de Physique des Particules, Centre National de la Recherche Scientifique). Institutions throughout the United States include Cornell Theory Center, Digital Equipment Corporation, Geophysical Institute at University of Alaska, Hewlett Packard, International Business Machines, Lawrence Livermore National Laboratories, Los Alamos National Laboratories, National Aeronautics and Space Administration (Goddard Space Flight Center), National Institute of Standards and Technology, Princeton Plasma Physics Laboratory, Sandia National Laboratories, Silicon Graphics Corporation, The Portland Group, United States Department of Energy, University of Colorado, and University of Illinois at Urbana-Champagne. Additionally, numerous international and domestic laboratories and institutions have indirectly acquired information over the Internet.

This growing level of attention, supported in part through activities of members from the examining committee and our article in Communications of the ACM [46], shows the interest and need for research of this kind. Therefore, thesis has been written for a wide audience, containing both a research and a tutorial perspective

where appropriate. This work should be readily accessible to scientific application developers whose background involves fields outside of computer science.

Although we explore paradigm issues in terms of scientific computation in plasma simulation, the points addressed easily extend to broader areas of interest. Our focus on physics-related issues is limited; rather we concentrate on aspects of simulation that are affected by these issues. In most circumstances, the topics addressed will be common to most scientific applications. Hopefully, you will find the details and conclusions drawn within this thesis suitable to your specific areas of interest as well.

To help guide your reading, we have included "briefs" at the beginning of each chapter. These are not summaries, but rather an overview of the issues discussed. Although there may be some temptation to skip various sections based on your curiosity, the most effective way to read this text is from the beginning. It is essential that you examine Chapter 1, since discussion of how this work fits into current events is only described therein.

This work builds upon itself relying upon information, assumptions, experiences, and insights revealed from previous chapters. In many instances this information is not repeated. The best way to benefit from this work is to observe the progression, development, and evolution of ideas and concepts presented. This should also make the reading experience more enjoyable and educational.

Finally, detailed knowledge of Fortran 77, Fortran 90, and C++ programming is not a prerequisite for understanding the contributions of this work. Developers interested in Fortran 90 and object-oriented programming will find a brand new approach to scientific programming where the traditional aspects of efficiency are combined with the abstraction modeling benefits of object-oriented technology. While experienced Fortran 77 developers may benefit the most from the techniques and lessons presented, those experienced in C++ should find the comparative analysis to Fortran 90 of interest. Experienced C++ programmers will find many of the modeling and development issues across compilers and machines enlightening. Additionally, the

work in object-based design paradigms for irregular computation will be of interest. Everyone considering language selection issues will find the sections on abstraction and the impact on object design in scientific programming useful, including the performance and language comparison sections. We hope that the analysis presented will inspire all readers to question and evaluate their development process, opening new design alternatives useful in modernizing their existing and future applications.

Charles D. Norton October, 1996

Troy, New York, U.S.A

Acknowledgments

There are a large number of very generous, intelligent and thoughtful individuals who have contributed to my research activities and lifestyle during my studies. Thinking back on my experiences, the complications and setbacks typically associated with graduate study are shadowed by many fond memories of the support and encouragement I have received from my family, colleagues and friends. I am pleased to introduce you to many of them through these acknowledgments, from which you may also learn something about me.

My advisor, Bolek Szymanski, is a very patient, insightful and energetic scientist with whom I have been privileged to conduct this research. His guidance and prudent timing have led to an unusual amount of exciting and enjoyable professional activities. In addition, Viktor Decyk has been extremely generous with his time and resources. His collaboration and clarity of thought has been thoroughly enjoyable and inspirational; I look forward to continued exploration of issues in scientific computing with Bolek and Viktor in the future. Joseph Flaherty, Franklin Luk and David Musser have been involved in my development long before they were officially part of the thesis committee. I deeply appreciate their on-going contributions to my studies.

Ephraim P. Glinert was instrumental in my graduate work, both as my M.S. advisor and in providing essential guidance during my transition to Ph.D level research. His support during this time was the catalyst without which my research goals might not have come to fruition. In addition to my family, committee and close friends, I dedicate this work to him.

Robert D. Ferraro, associate project manager of NASA HPCC, members of the High Performance Computing and Applications Group and the support staff at Jet Propulsion Laboratory, California Institute of Technology, always ensured that my numerous research visits were pleasant and productive. I have benefited considerably from their collaboration.

Bradley W. Dickinson, my former advisor from the Electrical Engineering Department at Princeton University, has followed my progress during my graduate work. Near the end of my B.S.E studies, he encouraged me to pursue the Ph.D and I am grateful for his foresight and continued interest in my professional development.

Gary Judd, Dean of the Faculty and Dean of the Graduate School at Rensselaer, immediately developed a keen interest in my academic career upon my arrival. Throughout my studies he has been a wonderful mentor and I appreciate his thoughtfulness.

Robert F. McNaughton has been a great friend and mentor, as he is to anyone fortunate enough to know him. Not only has he offered sound advice over the years, his appreciation and detailed knowledge of the fine arts, particularly the symphony and chamber music, is considerable. I have enjoyed many wonderful outings to Emma Willard for the "Friends of Chamber Music" series and the Saratoga Performance Arts Center with Robert and his wife Vivian.

Eva Ma, with whom I have also enjoyed many concerts, has been a constant source of encouragement. Whenever I had doubts about the progress of the research, an informal meeting quickly placed these concerns to rest.

While the entire Computer Science Department faculty has been unusually supportive of my efforts, I want to recognize Erich Kaltofen, Sam Kim, Mukkai Krishnamoorthy and Susan Rodger who clearly went out of their way to ensure the successful completion of this work.

The Computer Science Department laboratory and secretarial staff are an outstanding resource, without whom my research and travel activities could not have

been conducted. In particular, Joyce Brock, Dave Hollinger, Nathan Schmike, Ora Schongar, and especially Pam Paslow never hesitated to assist me. Additionally, the Numerically Intensive Computing Staff at Rensselaer and the staff at Jet Propulsion Laboratory, California Institute of Technology were extremely helpful; particularly Mike Kupferschmid, Nooshin Meshkaty, and Mark Miller.

My family has always believed in me, which is all that anyone ever needs. My Father passed away not long after I was born, so my Mother and Grandmother raised my brother and me alone. While they probably made a number of sacrifices during this time, I'm sure they would say otherwise. My Grandmother first encouraged my study of music and my Mother fully supported this through activities with various symphony orchestras. Those who know me well understand that music forms the foundation to most of my intellectual activities; hopefully I can continue to transfer the discipline, patience and respect for history the performing arts requires into my scientific research.

This thesis is dedicated to my Mother Alva, my brother Michael, my aunt Alice (who always supports major events in my life), and to the memory of my Father Jim, and my Grandmother Cleo. I cannot comment on what they mean to me, or the importance of their influence on my life.

I am fortunate to have many dear friends, however I want to bring special attention to some with whom I have enjoyed many pleasurable activities. Gail and Jon Berry, Chak Bhagvati, Steve Blythe, Patsy and Nui Bodhipaksha, Sandy Charette, William Chung, Ewa Deelman, Karen Devine, Maryanne Egan, Robin and Steinar Flatland, Rama Govindaraju, Varina Hammond, Monique Havasy, Wesley Kaplow, Susan Kokernak, Cynthia Lawton, Cassandra Lehman, Raymond Loy, William Maniatty, Julia Marvin, David McIntyre, Mohan Nibhanupudi, Austin Lobo, Toshiro Ohsumi, Kristina Ott, Susan Palmer, Spencer Phillips, Badri Ramamurthy, Patrica Ryan, Andrew Shapira, Balaram Sinharoy, Peter Tannenbaum, Kedar Tupil, Tom Valente, Rebecca Wells, Julie Yousefi, and Louis Ziantz. Should you come across any

of these individuals, introduce yourself and your life will be improved immediately.

My research has been sponsored by a variety of agencies and corporations and I wish to formally acknowledge their support. This includes the *General Electric Foundation*, *GTE Corporation*, *International Business Machines* and the *National Science Foundation* CCR-9216053. Most of the dissertation research was supported by the *National Aeronautical and Space Administration Graduate Student Researchers Program* NGT-70334 and carried out in part at the Jet Propulsion Laboratory, California Institute of Technology, and University of California at Los Angeles Department of Physics and Astronomy, in addition to Rensselaer Polytechnic Institute.

Support for international speaking engagements has come from *Laboratório de Cosmologia e Físca Experimental de Atlas Energias* (LAFEX), *FERMILAB* and *NASA Headquarters* for travel to Rio de Janeiro, Brazil in September of 1996. Additionally, the *University of Tokyo* (hosted by Dr. Satoshi Matsuoka), the *ACM SIGPLAN Professional Activities Committee* (chaired by Dr. Mary Beth Rosson) and an *IBM Development Grant* supported speaking engagements in Kanazawa and Tokyo, Japan in March of 1996.

Access to the Cray T3D and Intel Paragon XP/S at Jet Propulsion Laboratory, California Institute of Technology was provided by *NASA Office of Aeronautics*, *Office of Mission to Planet Earth* and *Office of Space Science*. Access to the IBM SP2 at Rensselaer was provided by the *Scientific Computation Research Center*. Additional access to equipment and software at the University of California was provided by the *UCLA Office of Academic Computing*.

Abstract

The prominent success of high performance computing used in modeling scientific, engineering, and physical phenomena continually motivates the development of very ambitious applications. In most instances programming languages such as Fortran 77 and C have been used, but many of the most challenging applications are stretching the capabilities of these languages. Many scientific problems are full of physical and mathematical abstractions, yet these are often difficult to integrate into the programming process. Furthermore, as modern languages and high performance computers are introduced, there is a need to explore their effectiveness, usability, and performance in scientific programming.

We have investigated traditional and modern object-oriented paradigms in scientific computation. This work is in collaboration with the National Aeronautical and Space Administration High Performance Computing and Communication (HPCC) Earth and Space Sciences Project at Jet Propulsion Laboratory, California Institute of Technology. The simulation testbed application used in our research is based on Fortran 77 plasma particle-in-cell (PIC) skeleton programs associated with the Numerical Tokamak Turbulence Project—an HPCC research effort to model and understand the transport of particles and energy in a tokamak fusion energy device. The plasma PIC model follows the trajectories of millions of particles in their self-consistent electromagnetic fields, both external and self-generated. Since the computation required is extremely large, both in terms of the field sizes and number of particles, the only alternative is to use massively parallel computers. While a variety of institutions are

involved in many aspects of this project, our efforts concentrate on the problems of evaluating new languages, machines, and programming techniques applicable to the next generation of advanced simulation programs.

Our contributions include the design and implementation of object-oriented plasma simulation codes using language-free abstractions, allowing analysis and comparison of language paradigms in Fortran 77, Fortran 90, and C++. This includes developing a methodology for object-oriented programming in Fortran 90, uncovering the new potential of the modern features provided by this language. Since advanced codes on parallel machines require dynamic load balancing, we have introduced an object-oriented instrumentation technique which monitors objects continuously and implicitly. This approach simplifies the introduction of load balancing extensions into object-oriented programs while preserving class hierarchies. Additional paradigm-related contributions study the implications of abstraction in scientific programming, with machine and compiler performance comparisons between Fortran 77, Fortran 90, and C++ on scalar workstations and high performance distributed memory parallel computers. This work advances our understanding of emerging language standards and new paradigms critical for modern scientific programming.

Chapter 1

Introduction and Historical Review

We introduce the importance of programming paradigms in modern scientific software development with a language independent overview of object-oriented methodology and concepts. This is followed by a brief review of the current state of the art in object technology for parallel scientific computation in plasma modeling, as well as commentary motivating the relevance and contributions of this work.

1.1 The Importance of Programming Paradigms

The effective programming of large scale scientific applications on massively parallel computers remains challenging. Although there are many concerns involved in building applications in this environment, the most significant include organization of the software for extension to new problems, clarity of design for shared development among multiple contributers, and the development of appropriate computational abstractions. These software engineering issues have become requirements in the management of increasingly ambitious scientific computations. This is particularly noticeable on supercomputers, which have traditionally been difficult to program.

The success of computer simulation continues to impact research in science and engineering, spawning new problems and increasing our understanding of various

phenomena. Nevertheless, these benefits are difficult to achieve and often result in complex, difficult to maintain stand alone programs, which lack the features needed to build multidisciplinary applications.

What is needed is a better understanding of how new programming paradigms can be applied to ambitious scientific applications.

> **Programming Paradigm**: A development methodology, not defined by a particular language, that characterizes the distinguishable way in which applications are designed and implemented.

Modern programming paradigms must consider irregular data and communication characteristics requiring dynamic load balancing, modular design for extensibility, and appropriate abstractions in programming. A programming language can simultaneously support development using a variety of paradigms. Similarly, paradigms take on different forms when parallel programming is of interest [57].

The demands of modern simulation are imposing limits on traditional programming approaches. New paradigms, which were largely ignored in scientific computing, are receiving attention and need to be critiqued. This is a critical time to evaluate past, present and future language paradigms since each user must make decisions, often based on a lack of complete comparative evidence. Fortran 77 programmers may have the most difficult decision, should they move into a completely different language like C++ or build upon their experience with Fortran 90? Migration to a new language introduces risks. Are the benefits of object-oriented programming in C++ worth the risk of losing large investments in expertise and existing software for the seasoned Fortran 77 programmer? What is the most effective way to program in Fortran 90? Many new features are available to support abstraction; how are they best introduced? Many computational scientists have migrated to C++, but are they receiving any benefits compared to their previous experiences in Fortran 77 and C, or are they losing ground due to the immaturity of compilers on advanced parallel architectures? Even if they are programming in C++, to what extent are they putting the

object-oriented methodology into practice, have they committed to a complex language without the background in object-oriented design to make this effective? What about users who have been programming in C++ and are unsatisfied with the performance of their codes? Should they consider moving to Fortran 90, or are the modern features of C++ (meta-classes, exception handling, expression templates, Standard Template Library) sufficient motivation to remain with this language? Should similar features be included in future revisions of the Fortran 90 language?

C++ programmers enjoy a powerful language. Advanced tools finding their way into software systems, particularly in the areas of visualization and analysis tools, work very well within the C++ framework. C++ is probably the fastest growing language among programmers; is there any real reason why other languages should even be considered in scientific programming? Finally, scientists who must move into the area of simulation but have limited experience with software development, often rely on rumors regarding language selection. What factors should one consider when choosing between traditional Fortran 77, Fortran 90, or C++ in scientific programming?

The general theme of this research involves empirical study of object-oriented programming paradigms for scientific computing. This has been indentified by the Grand Challenges Applications and Software Technology Programming Paradigms Working Group as a serious open issue: "(that) could be used to drive the development of the next generation of computer software [57]." The members warn that understanding the fundamental questions of the costs, benefits, and consequences of using one language rather than another can only be of interest if serious applications are considered.

The scientific computing community is looking for techniques to modernize and simplify development of advanced applications, particularly for extension to new problems and integration of related codes into multidisciplinary applications. For example, the National Aeronautical and Space Administration (NASA) High Performance

Computing and Communications (HPCC) Earth and Space Sciences (ESS) Project is a special program designed to advance the state of software technology, algorithms, and high performance computing for ESS problems related to NASA missions. One of the many areas targeted for research involves the study of parallel programming paradigms and dynamic load balancing aimed at advancing the state of software technology applicable to Grand Challenge problems [48]. Our research involves the analysis of various techniques using a plasma particle-in-cell simulation code to understand how modern paradigms can add structure and organization to computations of this kind. Additionally, by comparing various design approaches across machines, languages and development paradigms, a better understanding of the needs and challenges for current and future simulation programming becomes apparent.

Increasing the level of abstraction in scientific programming allows scientists to use many of the same concepts that have been beneficial in scientific theory. Additionally, a communication context is established which promotes interdisciplinary collaborations. These issues represent the importance of paradigm research.

1.2 Basic Concepts in Object-Oriented Methodology and Programming

Much of object-oriented methodology and programming is filled with terminology that can be difficult to grasp without extended examples. Furthermore, there are many research activities adding to this complexity on a regular basis. While debate continues on what makes a *specific language* object-oriented, this section briefly introduces the general concepts of object-oriented technology in a language independent manner.

> **The Goal of Object-Oriented Technology**: To facilitate the development of software from an application oriented viewpoint.

The underlying premise is that modeling software based on "real world" concepts enhance program safety, portability, modifiability and understanding. Since object-oriented programs can be represented through graphical class interaction diagrams, communication with collaborators, software readability, and documentation are improved.

The methodology emphasizes the design of conceptual models representing the problem, rather than language specific or implementation details. Many organizational approaches have been suggested; generally they require *analysis* of the problem domain, organization of the general *system design*, realization of system data structures and algorithm components through *object design*, followed by *implementation* in a specific language [55]. The main benefit of this approach is the clear distinction among development phases, useful in supporting large software development efforts.

There are many—often conflicting definitions—of what object-oriented languages must support. Nevertheless, the most widely used descriptions include at least the following [61]:

1. *Abstraction*

   The ability to represent concepts in a static form (classes) and to manipulate them in a dynamic form (objects).

2. *Inheritance*

   The ability to create new abstractions by preserving features of existing abstractions.

3. *Polymorphism*

   The ability for objects to share a single interface, while responding to operations differently, based on dynamic binding to specific routines.

Objects represent a unification of data with associated operations. These operations (often called methods or functions) modify the data encapsulated within the object. *Encapsulation* localizes where object definitions can be modified to well specified parts

of the program. A related notion is *information hiding* which defines the interface and access restrictions on the object. The interface functions may be publicly accessible or private. The combination of encapsulation and information hiding allows interfaces to remain fixed, even if the internal details of objects are modified.

An object is created from a *class*. Classes represents the main data structures of the abstract application model. When new classes share features in common with existing classes, *inheritance* allows usage of the base class features in the newly defined derived class. This represents one form of code reuse which is encouraged in the object-oriented paradigm.

Polymorphism allows objects to respond to a function without explicit knowlege of the object type. This can be very useful in generic programming. Suppose we have classes representing the sections of a symphony orchestra (strings, winds, brass and percussion) with objects representing various instruments such as violin, bassoon, horn, and tympani respectively. Objects of these classes may have a `tune` operation, used to verify intonation among other instruments in their section and the orchestra as a whole. While these objects share the same interface for intonation, they will respond differently. Violinists tune by moving a bow across the strings, wind and brass instrumentalists blow into various reeds or mouthpieces, while the percussionists often strike their instruments with a mallet.

Two additional language specific features are so often associated with object-oriented programming that we briefly introduce them here. These are *overloading* (compile time selection of a routine based on the number and type of its arguments) and *templates* (the instantiation of classes based on parameterized types).

Overview of Object Modeling Technique (OMT) Notation The Object Modeling Technique (OMT) notation [55] is part of the OMT methodology used in constructing object models. The general design methodology was not applied in our research, nevertheless the symbolic notation is helpful in representing our class hier-

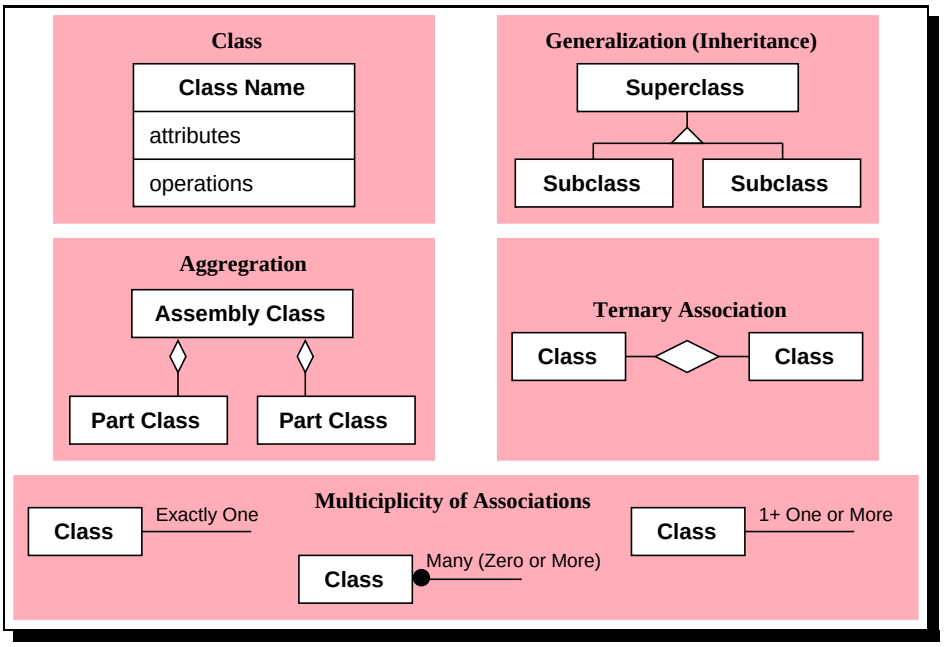


FIGURE 1.1: Basic elements of the OMT Notation.

archies. Figure 1.1 shows some of the basic elements of this diagramming technique. Most components are self explanatory. The links connecting classes show a one-to-one relationship, while numerical annotations indicate the number of object instances from that class. Links may contain *link associations* describing the meaning of the relationship between classes. *Link attributes*, when present, describe the property of the link. The diamonds attached to classes indicate an *aggregation* defining a "part-of" relationship. Aggregations show how an object is assembled from many component objects. The large diamond is a *ternary association* indicating classes which are so tightly related that they form an atomic unit, one cannot be fully described without the other. The examples in Chapter 3 will clarify all of these concepts.

Minor Commentary and Personal Viewpoints While object-oriented methodology is language independent, experience shows that *the actual construction of software is often influenced by language specific details.* Many object-oriented languages (and languages that support object-oriented programming) have been developed.

These languages support the methodology in varying ways and extent. *The definition of what makes a program object-oriented (or not) should never be based solely upon the implementation language features.* Constructs including overloading, C++ virtual functions, Fortran 90 array operations, and so on, only enable the methodology to be applied. These features must be used judiciously; very good object-oriented programs can be written without applying every construct. Rather, the ability to manipulate, modify, extend, share, and understand the software through abstractions determines the "object-orientedness" of the program. These features are in agreement with the goals of the methodology, not a specific language or feature implementation. Finally, *object-oriented techniques are not always useful for every kind of application.* The methodology should be applied when appropriate, which is more difficult to determine than when it is not appropriate! Only experience and honest evaluation during development can make this distinction clear.

1.3 Background and Relevance to Previous Work

Many researchers are investigating the use of object-oriented technology in parallel scientific computation [62]. These efforts focus almost exclusively on creating new programming languages, compilers, or extensions to support parallelism based on C++ or similar languages. Currently, the most promising technique for parallel programming combines a standard high-level language with an explicit message-passing library for interprocessor communication. Languages, however, can also be extended with new constructs in direct support of parallelism. The principal explicitly parallel Fortran-based language is High Performance Fortran (HPF) [23], which introduces new keywords and extrinsics for data placement and alignment. Additional research languages include Fortran D [24], Fortran 90D/HPF [4], Fortran M [16], Opus and Vienna Fortran. Some of these languages support operations on virtual processors which separates the problem partitioning and mapping from the physical proces-

sors. Research activities in object-oriented parallel languages include ACT++, C**, Charm++ [29], Compositional C++, Concert [30], Concurrent Aggregates, Concurrent C++, COOL [39], DC++, DCE++, HPC++ [63], Mentat [19], Parallel C++, pC++ [2], POOL-T, μC++ and many more. Many of these systems have been recently described in a new text [71]. 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. Additionally, libraries featuring VLSI CAD applications [49], finite-element/finite-volume computations (DIME++) [70] and materials science (LPARX) [31] have been developed. Generally, library-based approaches try to preserve existing C++ codes rather than introducing new languages or language extensions.

Although these efforts are important, the focus of our research involves the appropriate usage of object-oriented paradigms for scientific computation, not the extension of existing languages with features to support parallelism. 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 to adopting new languages. We believe development within existing and emerging standards is most useful for the widespread use of object methodologies in scientific programming. In this way, investigating the features of Fortran 90 and C++ is very relevant. New standards under development, such as HPF and HPC++ based on Fortran 90 and C++ respectively, will undoubtably receive increasing attention.

Object-oriented methods are finding their way into plasma simulation, although the current number of efforts is limited. Forslund [15] has performed simulations on a

network of workstations using C++ and the ISIS system. Objects are introduced to represent the particles and fields. Objects of the particle class know how to respond to electric and magnetic fields; particles are vectors. Objects of the field contain grid quantities with interpolation methods to and from the grid. This provides a vector field over the simulation space. A region class represents a process containing particles and fields on each processor. The boundary class describes the geometry of regions while a species class maintains particle properties. Class methods for advancing particles overload operators giving a program structure that closely matches the physical equations. Performance is mentioned as a problem with this system, based on the latency of communication among workstations. The system was scheduled for an upgrade to the Thinking Machines CM-2 parallel computer.

Haney [22] also uses C++ with building block classes of lists, queues, iterators and matrix and vector templates. Their tokamak modeling code is 70% C++ while the rest is in Fortran 77. The development structure is organized around the building block classes as opposed to specific physical constructs such as particles. This system was not developed for a high performance computing environment.

Reynders [50, 51] has developed an object-oriented particle simulation library which seems to continue the work of Forslund. The class structure has been reorganized and extended to handle abstractions for the distributed memory parallel computer and I/O operations. A patch object now contains grid data and the field consists of a group of patches which can be load balanced. A species object is analogous to the field object since it contains particle objects that can also be load balanced. Users interact with the field and species objects, the patch and particles objects are hidden. Field and species objects perform operations across all patches in the system. A configure object maintains information on the location of all patches. Current efforts are focused toward providing capabilities for moving whole patches (with related particles) across the system while maintaining the configure object. This capability is supposed to allow for abstract load balancing of inhomogeneous particle distributions.

Much of these efforts have found their way into the POOMA framework [71].

Verboncoeur [68] describes experiences in object-oriented PIC for modeling a microwave tube. The modeling abstractions include fields, grids, spatial regions, boundaries with particle group lists, and a particle group consisting of particle objects. Vector operations are supported to aid in representing the mathematical model. This system was not designed for a parallel computing environment, but according to personal communication this may change.

We have also, independently, developed an object model for plasma simulations from the existing Fortran 77 codes. Our abstractions realize that plasma simulation models inherently depend upon *interactions* between particles and fields [46, 47]. Our approach models this relationship from a physical and computational perspective. Chapter 3 covers the details of this effort with extended work in Chapters 4 and 5. This is relevant to previous research in the field since we introduce a new understanding of how the object-oriented paradigm can be used to design sophisticated scientific software based on existing and emerging language standards. Additionally, modeling for extension to problems in higher dimensions, design for modification without side effects, and the introduction of advanced features for load balancing—which maintain programming abstractions—are emphasized for this application. The language comparison and Fortran 90 parallel programming experiences have already influenced new considerations for the Numerical Tokamak Project [5], now The Numerical Tokamak Turbulence Project, in supporting extensions for advanced issues in plasma simulation on modern architectures.

1.4 Overview of Principal Contributions

As described in the preface and abstract, this research addresses object-oriented programming paradigms for scientific computation. Most of the contributed work is based on completely new approaches and ideas, rather than extensions from previous

work. This is summarized as follows.

- Design and implementation of object-oriented sequential and parallel plasma particle-in-cell programs, from the Fortran 77 versions, with analysis and comparision of organizational models using language-free abstractions.
- Establishment of a programming methodology for the new features of the Fortran 90 programming language based on object-oriented techniques.
- Analysis of the implications Fortran 77, Fortran 90, and C++ programming language paradigms impose on abstraction modeling in scientific programming.
- Development of a Fortran 90 abstraction modeling benchmark program which measures the performance effects of applying object-oriented features across compilers.
- Design and implementation of an optimal mapping algorithm used in managing irregularly slab-partitioned fields in load balancing.
- Creation of an object-oriented continous implicit monitoring techique that allows objects to be instrumented and monitored for dynamic load balancing while preserving class abstraction hierarchies.
- Performance comparisons of Fortran 77, Fortran 90, and C++ programming language paradigms for plasma particle-in-cell programs on scalar workstations and distributed memory parallel architectures.

We have received many comments from scientists in various organizations who have learned about some aspects of this research. For instance, Ian Wells of Hewlett Packard Massachusetts Language Lab—in regard to the our Fortran 90 comparative paradigm issues and the abstraction modeling benchmark software—had the following comment:

> You have done an excellent job of providing a focus point for the scientific community to quantify the tradeoffs between programming productivity and performance. This is a unique and timely contribution. I feel the performance side of decreasing the effort to produce software has (been downplayed) with all the recent hype over C++ and now Java. This benchmark will help out internal optimization groups, managers making decisions about containing development costs, and programmers who need to get maximum performance for the least effort. Excellent [69].

This statement generously summarizes of the scope and impact of the research contributions.

Chapter 2

Plasma PIC Simulation

This chapter introduces plasma particle-in-cell (PIC) simulation on scalar and distributed memory supercomputers. We examine the physical and computational features of the model, the simulation experiments performed, and the concepts and terminology used in this grand challenge application.

2.1 Overview of the Plasma PIC Model

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 electric 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 do not interact with each other directly, but through the fields which they produce. Particles can be located anywhere in the spatial domain, however, the field quantities are calculated on a fixed grid. In our example applications only the

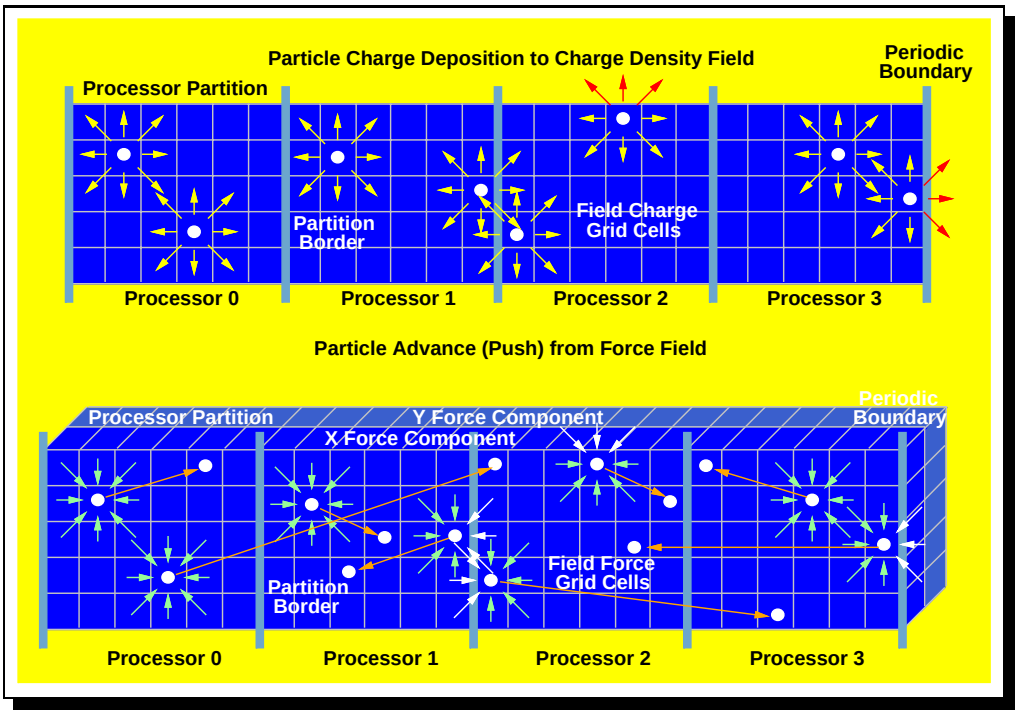


FIGURE 2.1: Particle/Field interaction in the plasma PIC algorithm (scalar and two-dimensonal vector field illustration): Local and non-local field and particle operations, due to the slab partitioning, are shown.

electrostatic (coulomb) interactions are included.

The General Concurrent Particle-in-Cell (GCPIC) Algorithm [34] partitions the particles and grid points among the processors of the MIMD (multiple-instruction, multiple-data) 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 2.1 showing the data dependencies that must be modeled in our class designs. Particles scatter charge and gather force data to/from their

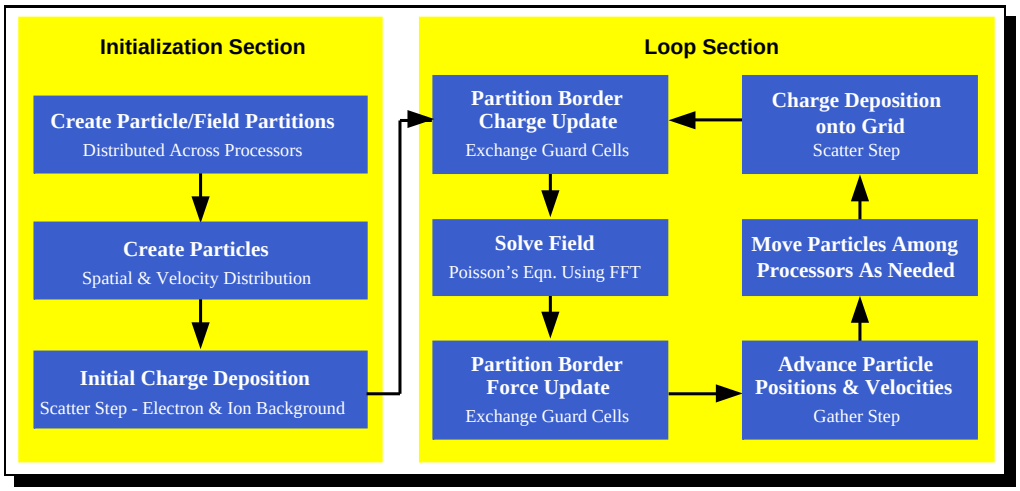


FIGURE 2.2: Plasma PIC computation loop overview: Diagnostic operations and extensions for load balancing are not shown.

nearest grid points. Electric field components from each dimension are required to advance particles to new positions.

The computation cycle, illustrated in figure 2.2, essentially consists of two major stages that occur with each simulated time step; the **particle push/advance** stage and the **field solve** stage (seen in the loop section). In the push stage, particle positions and velocities are updated using a time centered leap-frog integration scheme, based on the value of the fields at each time step. A near neighbor grid point interpolation method is used to find the plasma charge density from the particle positions. The electric field is then found by solving Poisson's Equation in Fourier space using the Fast Fourier Transform. Other field solution techniques can be applied, but the approach used is problem dependent. Diagnosics are computed along the way, where all lengths are normalized to the grid spacing. These can be related back to physical quantities later.

On a distributed-memory parallel computer, the GCPIC algorithm is written in SPMD (single-program, multiple-data) mode. In other words, every processor executes the same program independently, not the same instruction in synchronized lock-step typical of SIMD (single-instruction, multiple data) programs. The algorithm is

loosely synchronized by message passing and global communication operations. Since the fields and particles are partitioned, guard cells—areas of distributed data strutures that temporarily store data that belongs on a neighbor processor—are used to improve the efficiency of interprocessor communication. This allows many small messages to be transmitted as a single message, reducing message startup latency and other communication overheads.

Each processor is responsible for a region of physical space that includes fields and particles, as seen in figure 2.1 on page 15. Simulations that do not require dynamic load balancing of the particles and fields share identical partitionings across the processors. This is advantageous computationally since only the guard cell data need be transmitted between fields; this transmission is part of the algorithm and occurs every time step. When dynamic load balancing is required, the field and particle partitionings differ and may vary every time step. Since interpolation is required to map field quantities to the particles, and vice-versa, the GCPIC algorithm must be extendible to that case. This complexity is largely dependent on the numerical scheme used to solve for the fields. The modifications to the algorithm for load balancing will be presented in Chapter 5. Evaluating the use of object-oriented paradigms for such situations will be an important aspect of our investigation.

2.2 The Fortran 77 Simulation Programs

The Fortran 77 programs, which served as the basis of our research, were written by computational physicist Dr. Viktor K. Decyk from University of California at Los Angeles. These "skeleton programs," which contain the critical components of the main production programs, are used to explore the features of new computer architectures. Although very portable and well benchmarked, modernizing these programs to examine the features of new programming methodologies was of interest, making them ideal for our paradigm studies.

Some of the input parameters to these programs include the interval between successive time steps, number of iterations, system length parameters, number of particles in specific species and particle distribution function characteristics. The actual particles are generated internally with an initial density and velocity distribution. The output is a description of various plasma diagnostics along with performance measurements. The high level structure of the Fortran 77 programs is similar to the PIC computation loop in figure 2.2 on page 16.

The organization consists of two major sections referred to as the **initialization section** and the **loop section**. The initialization section builds the particle and field partitions (required for parallel programs), constructs tables, and performs the initial particle distribution and charge density deposition. The loop section calculates the electric field forces using Poisson's Equation and the Fast Fourier Transform, advances the particles under these forces, and finds the new charge density for the field at the grid points. When the particle and field partitions are distributed across processors, interprocessor communication is required to preserve consistency. Message passing libraries, such as MPI [20] or vendor specific libraries, may be used. Each loop represents a simulated time step during which diagnostics such as field, kinetic, and total energy are monitored.

A wide variety of constants, variables, data structures, and functions are used during the computation cycle. As mentioned, the parallel programs contain additional features associated with that environment; however, many fundamental characteristics are present in both the sequential and parallel simulations. Some of the most important simulation parameters are listed in table 2.1 on page 19. Although we show some components of the one-dimensional Fortran 77 sequential and parallel programs, the higher dimensional codes contain many additional data structures and variables. In fact, we have not included features associated with message passing or dynamic load balancing in table 2.1. These issues will be addressed later in Chapter 5. The programs were designed to separate the programming of physics related issues from

Sequential Parameters	Definition
part(1,n), part(2,n)	Array describing the position and velocity of particles in one dimension, respectively.
qme, qi0	Charge on electron and normalized Ion charge.
vtx, vtdx	Thermal velocity distribution parameters of background and beam particles.
vdx	Drift velocity distribution parameter of beam particles.
npx, npxb, np	Number of background and beam particles followed by the total number of particles.
nx	Simulation system length in x-dimension.
q(j), qc(j)	Real and complex charge density fields at grid point.
fx(j), fc(j)	Real and complex electric fields at grid point.
we, wke, wt	Field, kinetic & total energy diagnostics.
dt, nloop	Simulation time step interval and number of iterations.
Parallel Parameters	**Additional definitions required for parallel programs**
nvp, npp	Number of processors and total number of particles per processor in the current particle partition.
edges(2)	Array identifying location of field and particle partition boundaries on each processor.
nxp	Number of x-dimension grid points in partition.
kstrt	Processor identifier.

TABLE 2.1: Major simulation components of the one-dimensional Fortran 77 program. The sequential parameters are also used in the parallel programs. In higher dimensions, additional parameters are required.

those associated with parallel computation. This allows scientists that may not have experience with parallel programming to contribute to other portions of the codes.

2.2.1 The Sequential Programs

The Fortran 77 subroutines, with their parameters, are called in the initialization and loop sections with arguments from table 2.1. The most important routines of the one-dimensional sequential program are described in table 2.2. A sketch of the

Routine	`call distr1 (part,vtdx,vdx,npxb,idimp,nx)`
Action	**Initialize density profile and velocity distribution**. Given the particle array `part` and distribution parameters for the maxwellian velocity characteristic, distribute a specific particle species uniformly. Note that the system length and dimensions of the particle array must be passed as parameters.
Routine	`call dpost1 (part,q,qme,np,idimp,nx)`
Action	**Deposit charge**. The particles deposit charge, using a nearest grid point interpolation scheme, to the charge density field `q`. The number of particles, their charge and the system length are required.
Routine	`call push1 (part,fx,qtme,dt,wke,idimp,np,nx)`
Action	**Push particles**. The new particle positions and velocities are determined from the electric force field `fx`. The kinetic energy of the field is also computed.
Routine	`call pois1 (q,fx,isign,ffc,ax,affp,we,nx)`
Action	**Poisson solver**. The Poisson's Equation solver computes the electric field `fx` from the charge density array `q`, as well as the field energy.
Routine	`call fft1rx(q,t,isign,mixup,sct,indx,nx,nxh)` `call fft1rx(fx,t,isign,mixup,sct,indx,nx,nxh)`
Action	**Fast Fourier Transform**. The Fast Fourier Transform routine transforms the charge density array `q` to Fourier space for the Poisson solver and back to real space for the force field `fx` used by the `push1` routine.

TABLE 2.2: Fortran 77 major routines of the one-dimensional scalar program.

initialization section is shown in figure A.1 on page 190. Similarly, the loop section is sketched in figure A.2 on page 191. The organization of the one-dimensional programs is similar to the higher dimensional programs, although more complicated data structures are used.

2.2.2 The Parallel Programs

The parallel programs are highly portable data-parallel SPMD programs for distributed memory message passing computers. Although shared memory and vector parallel programs also exist, the distributed memory programs support larger experiments. Our attention will be focused on these programs.

The parallel programs are similar in structure to the sequential programs. Passing particles and field elements among processors represents the major difference between them, although various numerical operations also have modifications. For instance, the Fast Fourier Transform must now operate on distributed data, where distributed matrix transposes are applied during the transform in complex space. This impacts the organization of the Poisson solver since it now operates on distributed transposed data. As higher dimensional codes are written, the infrastructure and algorithms to manage the simulation become more complex. To illustrate some of the modifications introduced, we show the major subroutines of the one-dimensional parallel program in table 2.3 on page 22.

Most of the interprocessor communication library routines do not use buffers in transporting data. Rather, the Fortran 77 arrays are read and written directly as arguments to the routines. In higher dimensions, the functions shown are more complicated, but the general activites performed by the routines are similar.

Routine	`call pistr1 (part,edges,npp,nps,vtdx,vdx,npxb,nx,idimp, npmax,idps)`
Action	**Initialize density profile and velocity distribution**. The array `part`, partitioned by `edges` stores particles.
Routine	`call pdost1 (part,q,npp,noff,qme,idimp,npmax,nxpmx)`
Action	**Deposit charge**. The particles deposit charge to their local portion of the charge density field `q`.
Routine	`call ppush1 (part,fx,npp,noff,qtme,dt,wke,idimp,npmax, nxpmx)`
Action	**Push particles**. Some particles are moved to non-local field regions. Communication is performed in the `pmove1` routine. The kinetic energy of the field is computed.
Routine	`call ppois1 (qc,fc,isign,ffc,ax,affp,we,nx,kstrt,kxp)`
Action	**Poisson solver**. The Poisson's Equation solver computes the electric field `fc` in complex transpose space and the field energy.
Routine	`call pfft1r (qc,fc,isign,mixup,sct,indx,kstrt,kxp)` `call pfft1r (fc,qc,isign,mixup,sct,indx,kstrt,kxp)`
Action	**Fast Fourier Transform**. The distributed Fast Fourier Transform routine operates in complex fourier space using matrix transposition.
Routine	`call dcomp1 (edges,nxp,noff,nx,kstrt,nvp,idps)`
Action	**Partition region**. Compute partitioning of particle and field coordinates, stored in `edges` array.
Routine	`call pmove1 (part,edges,npp,sbufr,sbufl,rbufr,rbufl, ihole,jsr,jsl,jss,nx,kstrt,nvp,idimp,npmax,idps,nbmax, ntmax,ierr)`
Action	**Particle mover**. Move particles to the processor that owns the spatial region defined by `edges`.
Routine	`call cppfp1 (q,qc,isign,scr,kstrt,nvp,nxpmx,kxp,idps)` `call cppfp1 (fx,fc,isign,scr,kstrt,nvp,nxpmx,kxp,idps)`
Action	**Update guard cells**. Move guard cell data from particle partition to field partition and vice-versa. Charge density and force field data updated.

TABLE 2.3: Fortran 77 major routines of the one-dimensional parallel program.

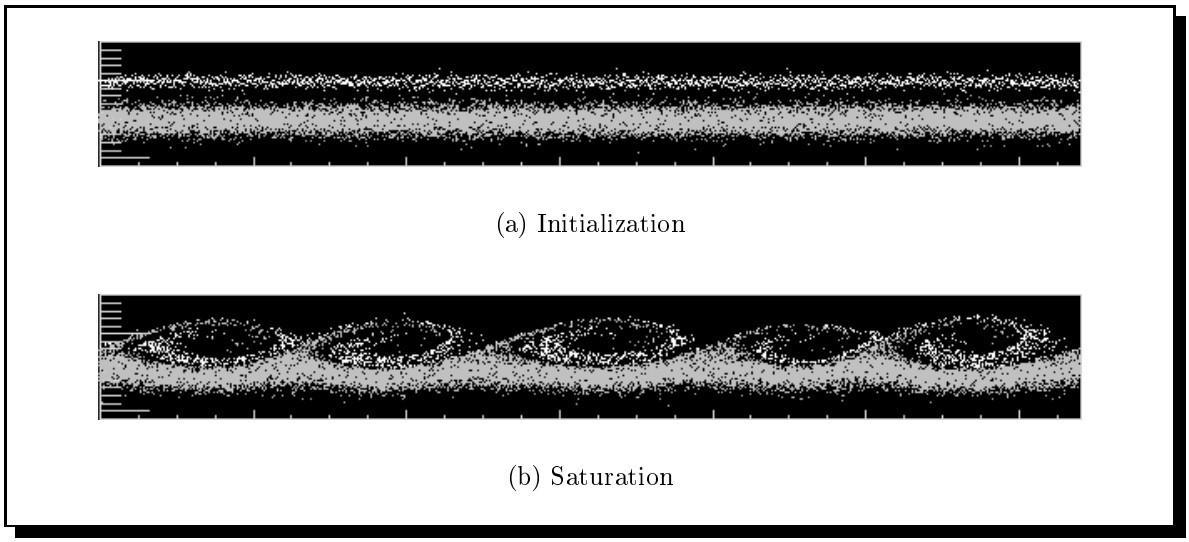
2.3 The Experiments

The plasma PIC method can support a variety of experiments based on the simulation parameters. Dimensionless units are used for computation since this allows the results of a particular experiment to be applied to similar cases without requiring additional simulation runs. The experiments we consider are electrostatic in a simulation region with periodic boundries. The kind of experiment performed is based on the physical properties of interest. Plasmas can be studied as a fluid, which is useful for long time scale experiments, but in the following we represent the plasma as a collection of particles. In all of our examples, we model charge-neutral plasmas—the collection of electrons and ions have a total net charge of zero.

2.3.1 Beam-Plasma Instability

The Beam-Plasma instability experiment models the injection of a low density electron beam into a stationary (yet mobile) background plasma of high density, driving plasma waves to instability. This is a special case of the more general two-stream instability where two streams of electrons flow through each other [25]. Beam-Plasma interactions cause particle bunching, forming potential wells which 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 [12]. 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 initial state of a one-dimensional experiment is shown in figure 2.3, where the phase space diagrams plot position against velocity. The instability drives plasma waves to saturation. In this experiment, we only measure the energy diagnostic



(a) Initialization

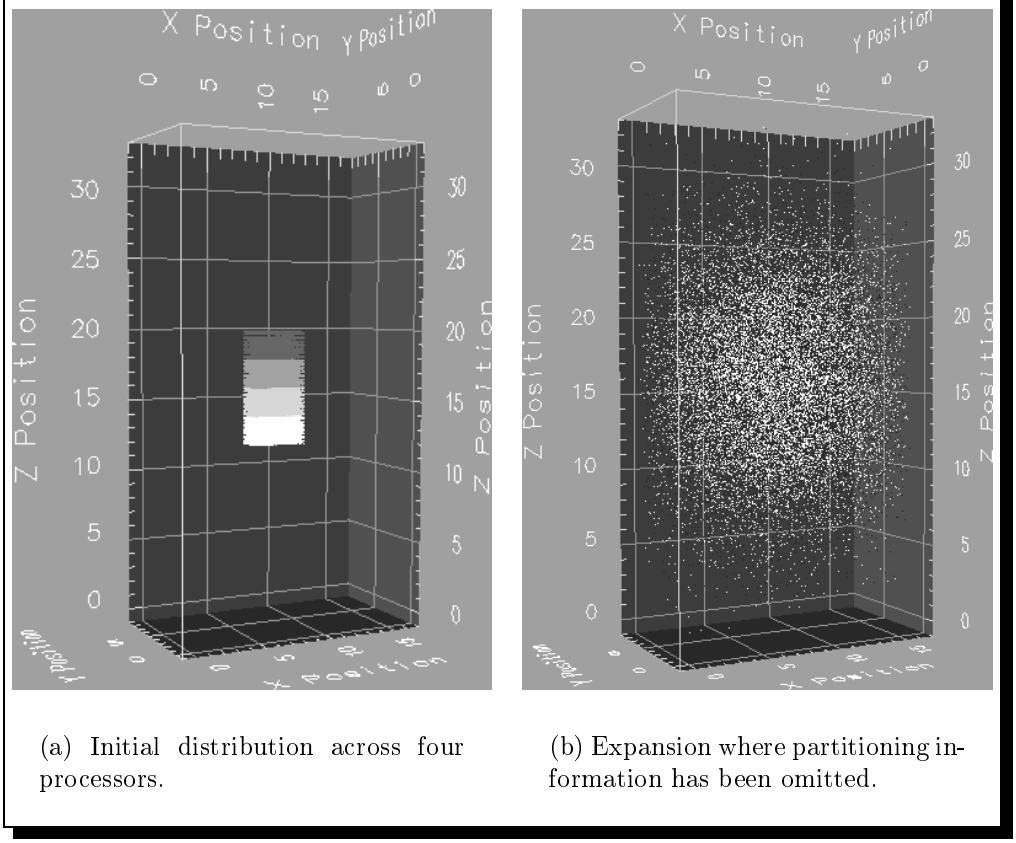
(b) Saturation

FIGURE 2.3: Electron phase space of one-dimensional beam-plasma instability.

which shows how the kinetic energy is rapidly converted into field energy.

2.3.2 Collisionless Free-Expansion into a Vacuum

The collisionless free-expansion of plasma has important applications in laser fusion experiments [10]. In this experiment, we allow a charge neutral plasma of electrons and ions to expand under its own forces. The electrons expand into the high density plasma regions as well as toward the vacuum region, which causes motion in the ions. Since the electrons and ions expand into the vacuum region, based on the electron/ion mass ratio, this experiment will allow for testing of dynamic load balancing techniques. Additionally, we can also evaluate how useful object-oriented methods may be for extending simulation codes to new problems, including how to design class hierarchies to encourage these modifications. An illustration of the expansion is shown in figure 2.4. The electrons and ions are initial partitioned uniformly across processors (c.f. 2.4(a)), spreading into the vacuum region symmetrically (c.f. 2.4(b)).



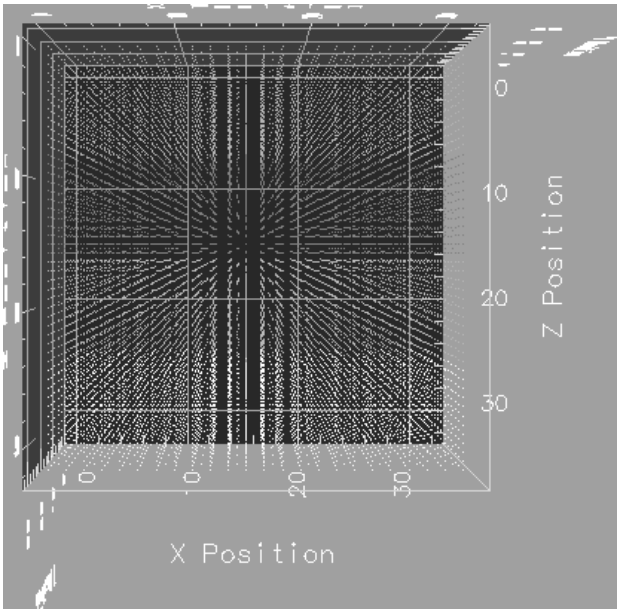
(a) Initial distribution across four processors.

(b) Expansion where partitioning information has been omitted.

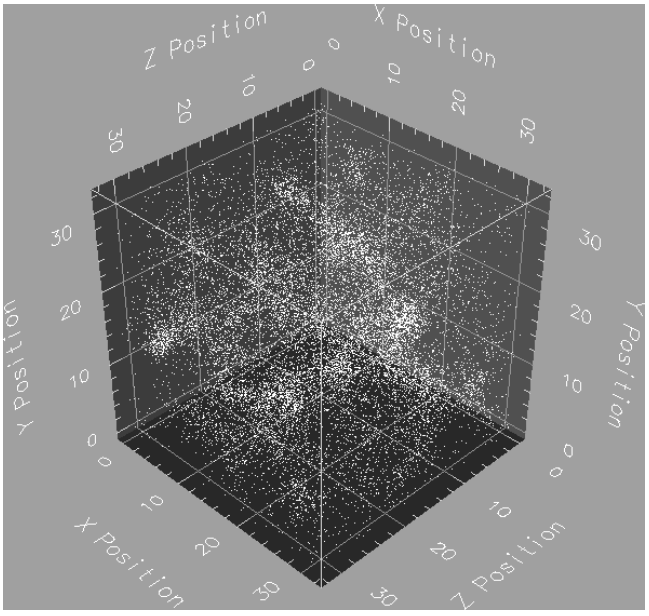
FIGURE 2.4: Electron-Ion collisionless free expansion into vacuum.

2.3.3 Gravitation

Gravititional problems can be addressed by using plasma PIC methods [25]. In terms of the simulation requirements, we can cause particles to bunch together by modifying Poisson's Equation and the normalization constant. The gravitational experiments will also provide nice examples for the study of load balancing and object-oriented design approaches. Figure 2.5 shows the initial uniform distribution of bodies in a periodic cubic space (c.f. 2.5(a)). As the simulation progresses, clustering of the bodies is introduced (c.f. 2.5(b)).



(a) Initial distribution across four processors (view from above).



(b) Development of clusters where partitioning information has been omitted.

FIGURE 2.5: Gravitational Bunching.

Chapter 3

Abstraction Modeling in Scientific Computing

This chapter covers the design and programming of the sequential and parallel object-oriented plasma PIC simulation codes in the C++ programming language. The abstract programming models, designed from the Fortran 77 programs, are described. This provides insight into paradigm-related issues relevant to converting procedural-based scientific codes into object-oriented models.

3.1 Organizing Object-Oriented Simulation Codes

The aim of object-oriented methodology is to model solutions to problems from the viewpoint of real world abstractions [55]. Abstractions provide concise representations of entities without concern for development-related details. They are essential to the overall design of systems, providing flexibility in organizational decision making. Frequently, however, the requirements of scientific computing extend beyond what is tangible. Many abstract concepts may not exist in a "real world" sense, although they have a mathematical reality all their own. Computational geometric grids, representing various fields, are one example we have already encountered. These

grids—while important in solving mathematical field equations—are a simulation abstraction modeling a real world effect, namely, the fields created by the presence of particles. Object-oriented modeling in scientific computation must intermix many views including real world concepts, simulation ideas, and mathematical abstractions. Indeed, the abstractions themselves may require a variety of views, such as collective or individual operations. The abstractions already inherent in the scientific problem must be represented during application programming, but how does one make this transition?

Many computational scientists are moving toward C++ for their software development since it supports abstraction in programming. Surprisingly, *research regarding the appropriate usage of object-oriented paradigms for scientific computing remained unaddressed.* Questions including the creation of proper abstractions in scientific computing, how to integrate abstractions to support simulation modeling, which paradigm issues are useful or inappropriate, and how scientific computations should be organized to take advantage of object-oriented techniques, seemingly were never researched during the paradigm-shift toward object programming in scientific computing. Training for application development unrelated to this area is common. However, in high performance scientific computing, the design and organization of object-oriented applications have very different requirements.

Various approaches have been presented for object-oriented plasma simulation problems [22, 50]. As part of our research involving the NASA High Performance Computing and Communications Earth and Space Sciences Project, we have written object-oriented programs from Fortran 77 codes. These programs are designed to investigate extensions to new advanced problems via this object paradigm. This chapter describes work using the C++ language, while Chapter 4 explores these issues in the context of object-oriented Fortran 90 programming.

3.2 Object-Oriented Design of Sequential PIC Programs

In Chapter 2 we examined the organization and major components of the Fortran 77 particle-in-cell programs. In these programs large arrays maintain particle and field data representing position, velocity, electric field, and charge density data. These arrays are passed by reference among functions which compute energy diagnostics at each simulated time step. Although passing data arrays leads to a very efficient simulation in Fortran, the data interrelationships are lost.

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. Particles with spatial and velocity distributions, fields with charge and force components, energy diagnostics, and collective operations all must be modeled to function within an object-oriented context. While the simulation algorithm—and its representation in Fortran 77—characterize many of the abstract operations performed, non object-oriented languages place less emphasis on the meaning of the data and more emphasis on its availability.

In object-oriented design, a classification of the data and associated operations emphasizes and reflects the meaning of the application. When redesigning a Fortran 77 program into an object-oriented structure, issues to consider include:

- The impact of Fortran 77 program structure on the design of classes.
- The interdependence between efficiency and class design.
- The appropriate usage of C++ features.

Although the Fortran 77 versions are well organized, establishing a relationship between the data and characteristic operations would be beneficial. There will be tradeoffs in identifying and organizing classes from a Fortran based code; this is discussed in development of the C++ programs from the Fortran 77 versions.

3.2.1 One-Dimensional PIC Simulation

Beginning with the one-dimensional Fortran 77 version, our initial objective was simply to redesign the PIC application in C++ using objects. The **Object Model Notation**[1] from the Object Modeling Technique (OMT) of Rumbaugh et al, will be used in representing our class diagrams [55]. Various diagrammatic methodologies exist, each of which has benefits and pitfalls—many do not accurately reflect issues which influence code development. Sometimes the model can be quite distinct from the software written, but this is a limitation of how diagramming methods can reflect something as powerful as a programming language. (The important issue of how language statements affect abstraction modeling and class design will be addressed in Chapter 6.) Our selection of OMT was made quite arbitrarily—the OMTtool [18] was available for drawing object diagrams. Nevertheless, we must emphasize that *the OMT methodology played absolutely no role in the design, modeling, or implementation of our class hierarchies except for the use of the OMT notation.* For clarity we minimize the use of the notation, showing classes with their attributes, operations, associations, and generalizations (inheritance). The details of class operations, when required for illustration and comparative purposes, will be described separately.

Analysis of the application and the Fortran 77 source indicates that classes should represent the field (computational grid),[2] particles (individually and collectively), and diagnostics. All components of the plasma simulation can be expressed in terms of these features. Organization the classes requires knowledge of their interaction and commonality. For instance, electrons and ions share the property that they are both particles described by their position and velocity. They can be organized into a class hierarchy from which common properties can be derived and distinguishing properties defined. Unfortunately this organization is insufficient—it only defines operations on

[1] The OMT notation methodology is summarized in Chapter 1.

[2] In this thesis field and grid may be used interchangeably, although the field may consist of many computational grid elements.

individual particles. We need the ability to operate on the collection of particles that make up a plasma, while simultaneously preserving individual particle operations. A plasma class can be introduced which moves and distributes these particle objects collectively.

Plasmas may also be viewed as a combination of particles and fields of various types, so this aggregation must be supported. A grid class may provide operations to deposit charge and solve Poisson's Equation for the electric field. An energy diagnostic class may interact with the grid class to collect information related to energy calculations. Additionally, classes which provide specialized services may be needed, such as random numbers or timing measurements. Our goal is to model these viewpoints while supporting aspects of the simulation environment.

The Initial Version The original one-dimensional Fortran 77 program simulated a beam-plasma instability experiment, as described in Chapter 2. In figure 3.1 we show the class modeling diagram for an initial implementation. 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, in C++, implies that the electron class has all the properties of the particle base class in addition to the specific features that define an electron. Since the beam-plasma instability experiment requires **Background** and **Beam** electrons, they were derived from the electron class as well. These particles, distinguished by their thermal and drift velocities, are defined as *static member constants* to conserve storage. Since these constants must be initialized on an individual basis as static class members in C++, they were not defined as part of the electron base class. This reasoning also explains why the charge was not included in the particle base class. Since many millions of particles may be created, using static initializers saves memory since only a single copy of the constant will be stored and known to all objects created from the class.

The **Grid** class provides operations to deposit charge and solve Poisson's Equation

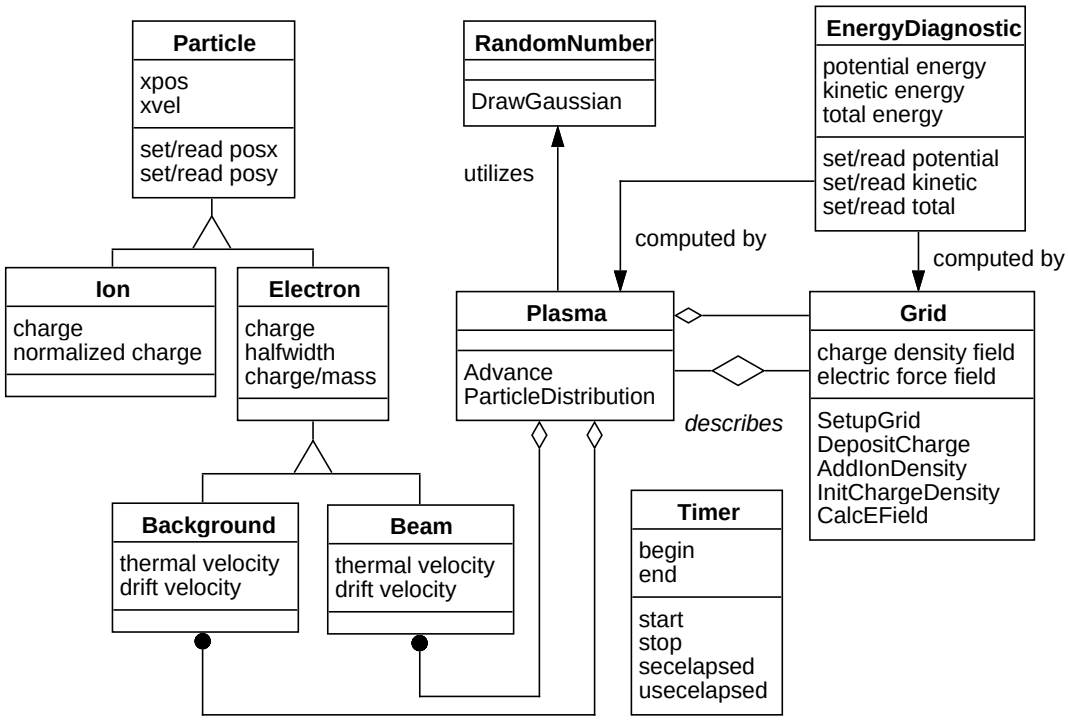


FIGURE 3.1: C++ hierarchy for initial version of one-dimensional scalar PIC code (OMT notation). The major encapsulated components and relationships are indicated.

for the electric field from the charge density field. It has two important attributes (components), the charge density field and the electric force field, which are arrays encapsulated into the grid class. The **Plasma** class contains no data, it only provides two functions that advance the particles and specify their initial spatial and density distribution. The **EnergyDiagnostic** class simply records this diagnostic during the computation. Finally, a class which provides Gaussian random numbers and a utility class for timing measurements are included.

Relationships among classes are also illustrated in figure 3.1. The plasma consists of an aggregation of particles and fields. Similarly, the plasma class utilizes the **RandomNumber** class, and the energy diagnostic is computed by the grid class. The arrows on these links indicate the one-way associations. The collective particles of the plasma conceptually form an alternative description of the physical field. Similarly,

the physical field describes the group of particles—they are distinct, yet bound to each other. This is a ternary association since particles and fields are atomic, one cannot be described without the other. Nevertheless, they also have very distinct properties. Incidentally, the ternary association involves the plasma class, and not the particle class, because the former represents the collection on which the association is based.

This hierarchy permits the software design to reflect the features of plasma simulation in general, and the beam-plasma instability experiment in particular. Although the example is small, many of the functions and routines of the Fortran 77 version in table 2.2 on page 20 have a new form, seen in table 3.1. Note that *the operations that modify the data associated with a concept are bound to that class.* This explains why charge deposition is a grid class operation and not a plasma class operation—charge deposition modifies the grid. Additionally, since the Poisson's Equation solver and FFT are private to the grid class they are not accessible by the main program so they are not listed in table 3.1. This is another example of encapsulation and information hiding.

The C++ object-oriented organization has advantages over the Fortran 77 version. The operations are performed on objects that represent simulation and physical abstractions. Also, the function argument lists are shorter due to encapsulation of simulation components. The main program is shown in figure A.5 on page 194.

Nevertheless, there are a number of disadvantages associated with this model. The background and beam particles exist in separate data structures causing operations to be called multiple times, such as `DepositCharge`. The difference between these particles is a property of their distribution characteristic, not their usage. Once distributed background and beam electrons are simply particles that share identical collective operations. This problem affects the computationally expensive `Advance` routine. Calling `Advance` twice only because the particles are stored in separate objects introduces performance penalties. Additionally, various constants such as NBMX (the number of beam electrons), are passed as parameters to object routines.

Objects	`plasma, grid, bkelec, bmelec, energy`
Call Syntax	`object.memberfunction()`
Definitions	The `plasma` object performs collective operations on the group of background (`bkelec`) and beam (`bmelec`) electrons while the `grid` object operates on the field; `energy` stores this diagnostic.
Routine	`plasma.DenVelDist(bkelec, BackgroundElectron::thmlvelx, BackgroundElectron::driftvelx, NBKX);` `plasma.DenVelDist(bmelec, BeamElectron::thmlvelx, BeamElectron::driftvelx, NBMX);`
Action	**Initialize density profile and velocity distribution**. Distribute background and beam electrons using the plasma object.
Routine	`grid.DepositCharge(bkelec, Electron::charge, NBKX);` `grid.DepositCharge(bmelec, Electron::charge, NBMX);`
Action	**Deposit charge**. Deposit charge onto grid from background and beam electrons.
Routine	`plasma.Advance(bkelec, grid, energy, NBKX);` `plasma.Advance(bmelec, grid, energy, NBMX);`
Action	**Push particles**. The plasma object advances particle positions and velocities using the grid object while assigning the potential energy.

TABLE 3.1: C++ major routines of the initial one-dimensional scalar program. Simulation objects and member function operations are indicated.

This information should be encapsulated as part of the distribution of particles.

Evaluation and revision are a necessary part of abstraction modeling. This is particularly important when concepts have multiple meanings. The design in figure 3.1 *models the Fortran 77 version too explicitly during the translation process to C++.* The revisions that follow take a more abstract view of the simulation process and the general meaning of its components.

First Revision The class hierarchy of figure 3.1 can be modified to include the distribution properties of the electrons within the **Electron** class. This eliminates

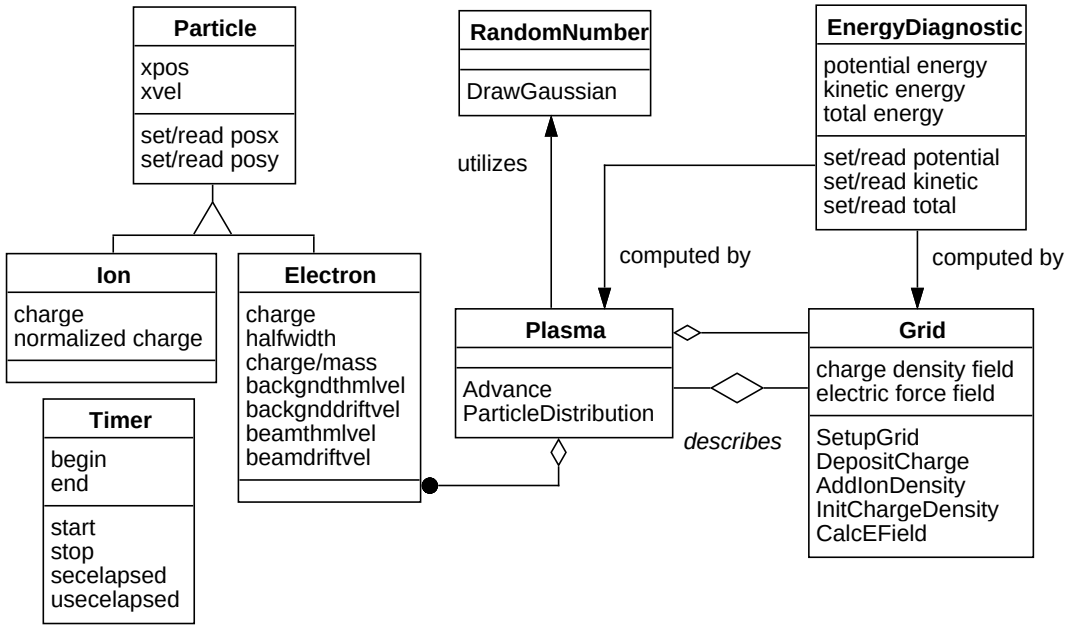


FIGURE 3.2: First revision of the C++ one-dimensional scalar class hierarchy (OMT notation). Electrons have been modified to include distribution properties.

the need for background and beam classes. Operations in the main program will remain identical to those in table 3.1, however; `DepositCharge` and `Advance` will only be called once on the collective group of electrons. The particle distribution routine necessarily must be called twice because the background and beam particles are distributed differently. Figure 3.2 shows the revised version of the class hierarchy which should be compared to figure 3.1 on page 32.

Notice how minor the class structure changes are, and how the modifications were limited to well-specified encapsulated components. Many routine parameters which expect background and beam electrons as particle types *require no modifications at all*. Of course, the benefits are more significant when larger and more sophisticated programs are involved. We will see examples of this in later sections.

Second Revision Modifying the electron class in figure 3.2 solved the problem of reducing function calls, but we must reconsider the effect of this decision. Earlier,

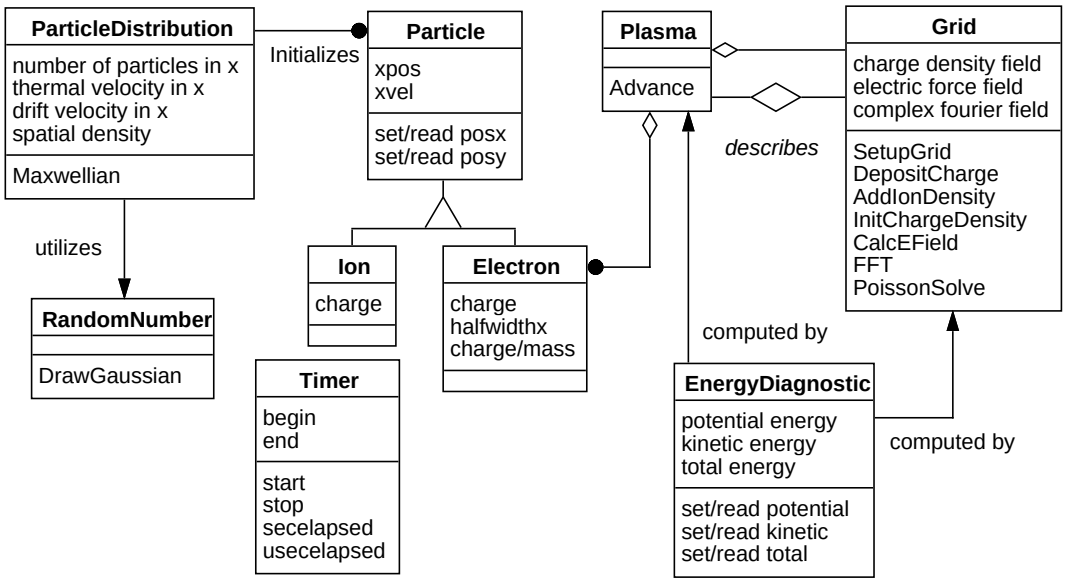


FIGURE 3.3: Second revision of the C++ one-dimensional scalar class hierarchy (OMT notation). Particle distribution properties are specified separately from the definition of particles by feature encapsulation into a `ParticleDistribution` class

we mentioned that thermal and drift velocity attributes specified the distribution properties of particles. These are simulation features, many possible distribution characteristics can be introduced. This mathematical property has no intrinsic relation to the physical properties of an electron. The distribution properties of particles should not be encapsulated in the electron class.

Figure 3.3 shows the class modeling diagram for the second revision of the one-dimensional sequential PIC program. A **ParticleDistribution** class, responsible for providing distribution features for particles, has been introduced. A distribution object *initializes* the electrons. Now, background and beam electrons may have different distribution properties without modifying the definition of an electron. An added benefit is that distribution operations, such as the spatial density for a particular particle species, are easily computed. Although we provide a Maxwellian distribution, additional functions can be added to this class in a manageable way.

Introducing the particle distribution class affected the plasma class, since the latter

was originally responsible for distributing particles. This caused a relocation of the random number class, but all of these modifications are safe due to encapsulation. A new association has been introduced between the particle distribution and the particle class—the former now initializes the spatial and density distribution of particles. The main program from the second revision is illustrated in figure A.6 on page 195.

Modeling changes in scientific computing may also affect how numerical computations are performed. In the grid class, we previously solved the Poisson's Equation in real space, but we moved to complex Fourier space for the FFT. As a result, we have added a new complex Fourier field to the grid class for this computation. The modifications were limited to the grid class operations, so users of the public interface for solving field equations were *unaware that the internal field solver routines have changed*. The interface remained fixed during these modifications for operations such as:

```
grid.CalcEField( energy );
grid.DepositCharge( elec, Electron::charge, NP );
```

when compared to the same operations previously defined in figure A.5 on page 194:

```
grid.CalcEField( energy );
grid.DepositCharge( bkelec, Electron::charge, N_BKELE_X );
grid.DepositCharge( bmelec, Electron::charge, N_BMELE_X );
```

Since the distribution objects specified how the collective electrons were distributed, only `elec` is needed as an argument to `DepositCharge`. Although the arguments changed the interface remained the same, because the background (`bkelec`), beam (`bmelec`), and electron (`elec`) objects share the particle type in the `DepositCharge` declaration statement:

```
void DepositCharge( const Particle elec[], const float qm,
                    const int np );
```

The organization of figure 3.3 seems satisfactory, but how flexible is it? In scientific computation software rarely remains unchanged—features and additions are added routinely. Evaluating the benefits of the object-oriented approach depends on examination of more complex problems, including extensions toward higher dimensional programs.

3.2.2 Two-Dimensional PIC Simulation

The beam-plasma instability experiment in two dimensions is similar to the one-dimensional case. Particles now have four components for position and velocity in x and y. Similarly, the particle distribution class now distributes particles into a two-dimensional geometry. The grid class contains multiple two-dimensional computational grids—one for the charge density field and one for each component of the electric field. Additionally, a complex field is included since the field solver operates in complex Fourier space. Figure 3.4 shows the organization, which should be compared to the one-dimensional version in figure 3.3 on page 36.

The main program for this hierarchy is identical to that of figure A.6 on page 195. One reasonable exception is that distribution objects in two dimensions require additional parameters (example for background distribution objects):

```
ParticleDist backelec( 1.0F, 0.0F, SYSLEN_X, N_BKELE_X );  // ONE-DIM
ParticleDist backelec( 1.0F, 1.0F, 0.0F, 0.0F, SYSLEN_X, SYSLEN_Y,
                       N_BKELE_X, N_BKELE_Y );             // TWO-DIM
```

The public interfaces are the same across the programs, but the *definition* of the objects has been extended to handle two-dimensional simulations.

3.2.3 Three-Dimensional PIC Simulation

When considering the design of a three-dimensional code there are clear similarities to the two-dimensional version in both the computation structure and program or-

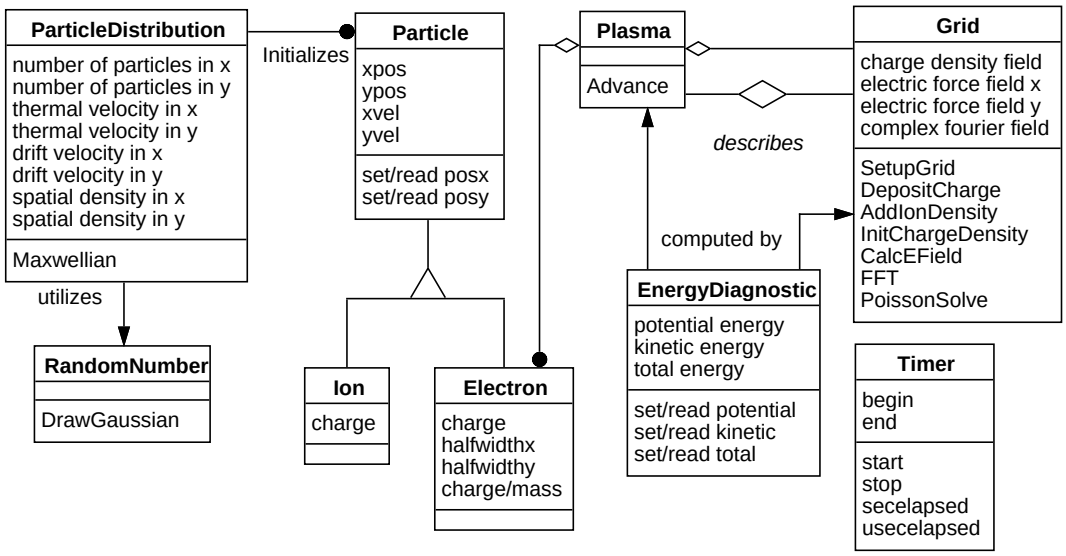


FIGURE 3.4: C++ two-dimensional scalar class hierarchy (OMT notation). Encapsulated components are extended while the interface to major routines remains unchanged. Object definitions reflect the higher dimensional problem.

ganization. Many of these similarities are extensions from encapsulated features of the lower-dimensional programs. Particles now have three components in x, y and z for position and velocity, an immediate extension from the two-dimensional case. Regarding the numerical aspects of the simulation, charge deposition—which used a 3-point stencil near-neighbor grid point scheme in one dimension and a 9-point stencil in two dimensions—now applies a 27-point stencil in three dimensions. This can also be viewed as an extension of existing abstractions.

Design for Extensibility The notion of *extensibility* is critical to abstraction modeling, particularly in scientific computation. Object-oriented designs may model a problem accurately, but if this model cannot be extended it may not be very useful. A major goal of this paradigm is not to reduce the effort of building the initial software, but rather to reduce the effort of producing software in the future. The initial design may be more difficult to create than more advanced versions, since a designer must account for general aspects of the problem with a viewpoint toward

future applications. A difficulty with the procedural programming paradigm is that software using this methodology is not usually designed to be reused. There is often little flexibility in the design of data structures or computational control preventing programmers from leveraging their existing work toward solving related problems. Good object-oriented models always consider how applications developed today can be leveraged toward those required tomorrow.

With these thoughts in mind, we must now consider some important questions regarding our previous programming efforts. How well were the previous codes designed for extension, both for new experiments and for higher-dimensional problems? Do our abstractions accurately allow for extensions in both the numerical aspects of the software design and the qualitative modeling of the physical problem? What about the simulation component; did we really consider how much effort would be involved in changing the features of a computational grid point for extension across simulation dimensions? We should point out, however, that many of these issues could not have been considered until we attempted to design a new code from an existing code. As new organizations are examined new questions and options appear. This is a natural aspect of this design process, which becomes more challenging when working with scientific codes, since simulation, mathematical, and abstraction modeling requirements must be satisfied simultaneously.

In the two-dimensional hierarchy of figure 3.4 on page 39 the particle class definition extended one-dimensional particles into two dimensions. Particles were also distinguished by their charge, using inheritance of basic particle properties into specific electrons and ions. There are mathematical operations involving particles which encourage a vector model definition. Also, particles potentially have fractional amounts of charge rather than simple positive, negative, or neutral charges. If we generalize the notion of a collection of particles to a vector space of charged particles, then vector operations on the collection can be supported.

We know that the field consists of a number of computational grids for the electric

and charge density fields. Based on how the field solver is defined, real and complex grid structures may be used. These fields share similar geometries and additional fields may be needed as new effects are added, such as current density for the presence of an externally applied magnetic field. A more general organization of the field could unify the components into a grid point, where collections of grid points form the fields in real and complex space. From an efficiency point of view, combining multiple field components for a specific point into a grid point structure can improve cache usage for expensive field operations, since components tend to be referenced together.

The Multi-Dimensional Vector Model Figure 3.5 shows the three-dimensional class hierarchy. Notice that the OMT notation has been extended to indicate classes which use the template feature of C++.[3] The modified class hierarchy uses templates to operate on a vector space of particles. A particle is generalized by a vector from the **PointVector3D<T>** class which represents the position or velocity components in the corresponding dimensions. This class defines mathematical point vector operations. This type is 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 instantiating a **Vector<ChargedParticle>** template class (using the **Vector<T>** template). This allows for vector operations on the collective group of particles. Note that the vector space of particles from the plasma class is specified using two separate objects:

```
Vector< ChargedParticle > elec_pos( NP );    // particle positions
Vector< ChargedParticle > elec_vel( NP );    // particle velocities
```

Since mathematical vector operations on particle velocities (such as scalar multiplication) must not influence position components, this distinction is necessary. This

[3]Templates allow specific classes to be generated by using object type information as parameters. The Standard Template Library[38] was not used since support was not available on our parallel machines.

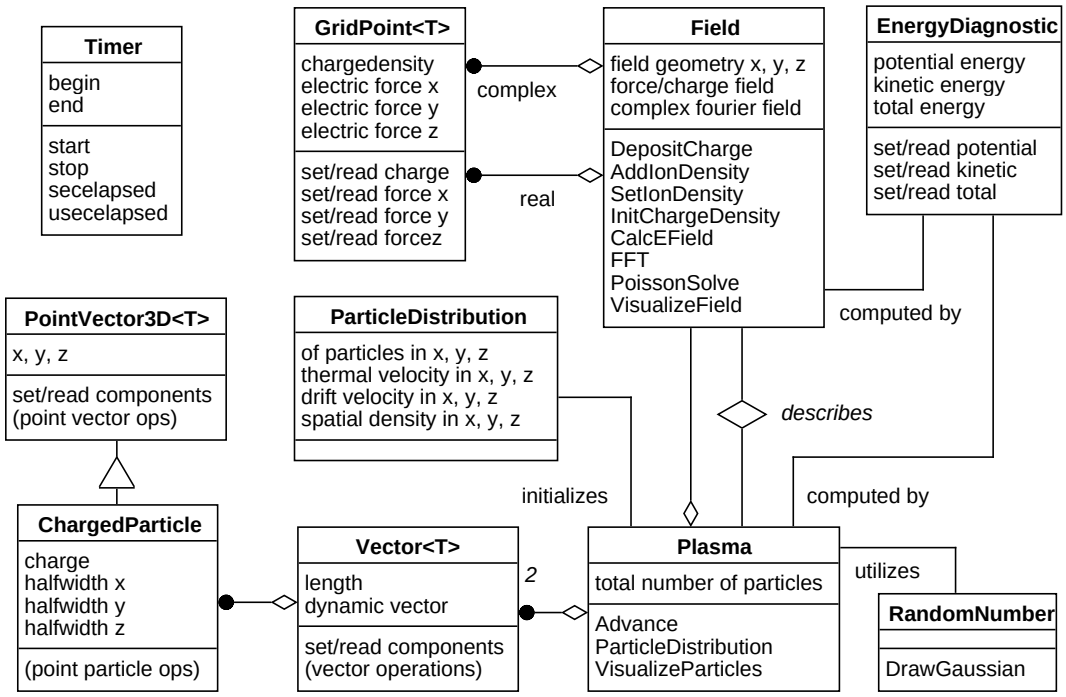


FIGURE 3.5: C++ three-dimensional scalar class hierarchy (OMT notation). Vector particle components with mathematical operations and parameterized grid point elements have been introduced.

class uses mathematical operations on entire vectors, which are propagated to specific point vector operations based on the dimensionality of the particle type. The **Field** consists of computational grid points of **GridPoint<T>** template class objects which unify force and charge data in multiple dimensions. Since there are real and complex fields, which vary in size, multiple aggregation associations are shown. The remaining components of the hierarchy perform the same functions as described in the previous models.

The main program for the three-dimensional hierarchy is shown in appendix A.7 on page 196. This should be compared to appendix A.6 on page 195. The programs are comparable; all of the vector operations have been information-hidden in class definitions. Since the particle distribution class specifies the *properties* of various distributions, such as background and beam electrons, the plasma class performs the actual distribution. A new function name reflects the uniform spatial and Maxwellian

velocity distribution performed. The multidimensional field is automatically initialized upon creation, so the `grid.Setup(energy)` call is no longer required.

The benefit of this new organization is that many of the features that vary across simulation dimensions are parameterized. Examples include the particles and the specification of the field components. Additionally, some of the mathematical operations—which must be modified with the dimensionality of the problem—are now standardized by vector operations that remain fixed. For instance, computing the new velocity for a particle, given the new particle acceleration, requires the following statements in Fortran 77 (additional program statements omitted):

```
c new velocity
      do 10 j = 1, nop
      dx = part(4,j) + qtm*dx
      dy = part(5,j) + qtm*dy
      dz = part(6,j) + qtm*dz
   10 continue
```

In C++, only one statement is needed due to the vector types (additional program statements omitted):

```
for ( register int j = 0; j < nop; j++ ) {
      diff = part_vel[ j ] + ( QTME * diff );   // New Velocity
}
```

In fact, if the `diff` acceleration components are known for *every* particle, we can modify the previous C++ statement so that indexing over a loop is not required:

```
diffv = part_vel + ( QTME * diffv );      // New Vector Velocities
```

where `diffv` is the vector of all new particle velocities. Given the large number of particles, the extra memory required to store the temporary vector could easily exceed storage limitations. Therefore, new velocity operations are performed on a

Objects	`plasma, field, elec_pos, elec_vel, energy`
Definitions	The `plasma` object performs collective operations on the group of particles with specific distribution properties (`backgnd`, `beam`). The `field` object performs field operations; `energy` stores this diagnostic.
Routine	`plasma.UniformSpcMaxwellVelDist(elec_pos, elec_vel, field, backgnd);` `plasma.UniformSpcMaxwellVelDist(elec_pos, elec_vel, field, beam);`
Action	**Initialize density profile and velocity distribution**. Distribute background and beam electrons using the plasma object.
Routine	`field.ChargeDeposition(elec_pos, plasma, ChargedParticle::e_charge);`
Action	**Deposit charge**. Deposit charge onto field based on particle positions.
Routine	`plasma.Advance(elec_pos, elec_vel, field, energy);`
Action	**Push particles**. The plasma object advances particle positions and velocities using the field object while assigning the potential energy.

TABLE 3.2: C++ major routines of the three-dimensional vector-based scalar program. Simulation objects and member function operations are indicated.

particle-by-particle basis. Nevertheless, statements such as these remain dimension independent.

In table 3.1 on page 34 we showed the major routines in the one-dimensional initial scalar program. Table 3.2 shows how these routines have been modified based on the three-dimensional vector organization of figure 3.5 on page 42.

3.2.4 Commentary on Sequential Models

Comparing figure 3.2 on page 35 to figure 3.5 on page 42 there are significant differences in the class designs. This raises a number of interesting questions. What effect does an existing Fortran 77 program have on the design of an object-oriented version

of that program? When considering the move to C++ from a Fortran 77 background this may be an issue in the decision if existing software must be converted. How does efficiency affect class design and how do we determine which C++ features are appropriate?

Generally, *the Fortran 77 source code did not influence many issues in C++ programming. Understanding the nature of the problem and designing the appropriate objects was much more important.* Some fundamental concerns included Fortran 77 array indexing. We preserved standard C++ conventions, indexing from zero, to prevent potential array indexing errors during source code translation. If Fortran 77 numerical routines were used through the foreign language interface to C++, Fortran-based indexing would have been prefered. We found, however, that many popular compilers had serious bugs so multi-paradigm language constructs were not introduced. Fortran 77 common blocks and equivalence statements introduced additional considerations. Fields in real and complex space were often equivalenced in Fortran 77 for the Fourier Transforms, saving memory. C++ does not support an equivalence construct, so additional data structures were required to store fields in complex space when complex transforms were applied.

Efficiency concerns can influence class design, particularly when a language like C++ provides features for improving efficiency. The Fortran 77 force fields were stored in very large independent arrays, where a component from each array was required in particle position and velocity modifications. In C++, by encapsulating each component into a single structure, referencing components does not require jumping between multiple arrays, which improves performance. Certain features of C++, such as virtual functions powerful for generic programming, were not required in the sequential programs. Some particle programs use pointer lists of various particle types with virtual functions operations. Background and beam particles could have been maintained in pointer lists, where virtual functions could distribute them based on their distinct type. Our simulations may use many millions of particles, however, so

pointer lists would waste memory prohibitively. Nevertheless, using virtual functions would have been inappropriate since other language techniques could have been applied with greater efficiency. Such issues demonstrate how class designs and efficiency issues are related, but this will be investigated further in Chapter 6.

Although the sequential object-oriented models differ from each other, where the three-dimensional version is the most general, they capture the essence of the simulation in a form that is more suitable for modification and extension. These differences are often in the inner details; the outer interfaces remained relatively unchanged. Modeling a scientific code involves a mixture of simulation, physical, and computational issues which interact with each other. In the next section, the effects of parallelism will be discussed.

3.3 Object-Oriented Design of Parallel PIC Programs

The sequential plasma simulation programs are useful for small scale experiments, and more importantly, for testing concepts before they are introduced into the larger parallel PIC codes. Since simulations with very large numbers of particles are of interest, the only option available is to develop software using massively parallel computers. Programming in a parallel environment adds additional complexity to an already complex problem. Returning to figure 2.1 on page 15, the fields and particles are partitioned across processors requiring interprocessor communication to manage data and numerical operations. Figure 2.2 on page 16 illustrates the modifications required to support parallelism. Field borders must be exchanged between processors and particles must be redistributed as they move across partitions. The partitioning remains fixed for problems that do not require dynamic load balancing, Various complications arise, however, when load balancing is required. Programs that do not require moving partition boundaries will be discussed in this section while the

modifications for load balancing will be addressed in Chapter 5.

Our sequential program design experiences will be beneficial. Some of the one-dimensional and two-dimensional parallel programs were developed simultaneously with the sequential versions. Nevertheless, we will ignore the actual order of code development to improve the clarity of this discussion. Furthermore, since many alternative code designs were examined, using approaches similar to the sequential codes, we will limit our discussion to the most recent versions. Issues involving how objects are designed for parallel programming will be considered after the models have been presented.

3.3.1 One-Dimensional PIC Simulation

The parallel codes must account for message passing of particle and field data across processors. Since each processor is responsible for a region of space, which includes a portion of the partitioned field and particles, constructs to define the properties of a region and the communication mechanisms to move data across regions are required. When designing the three-dimensional sequential code, we introduced templates and vector spaces to parameterize various features for extension. We plan to reuse this effort in building a one-dimensional parallel code.

Figure 3.6 shows the class hierarchy for the one-dimensional parallel program. Although rearranged, it compares well with the three-dimensional sequential version previously described. A **VirtualParallelMachine** class encapsulates machine specific communication information for portability among various architectures. The message passing operations allow objects to be transported among processors through a standard interface to machine specific message passing libraries.[4]

A **PartitionRegion** class partitions the fields and particles across the distributed memory machine. Link attributes describe partition association properties where a

[4]Much of the interface to the message passing operations was removed when MPI became available on our parallel machines.

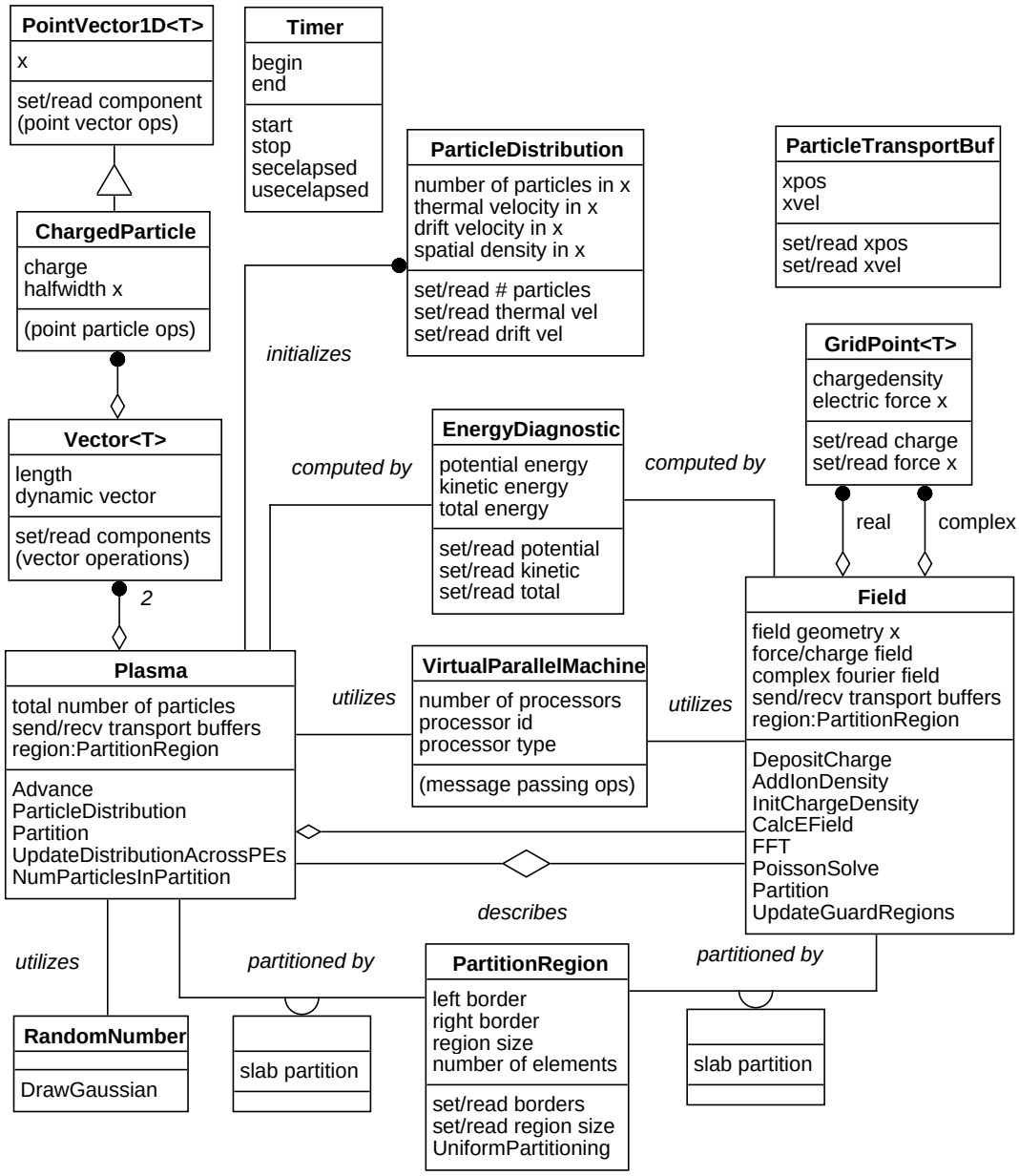


FIGURE 3.6: C++ one-dimensional parallel class hierarchy (OMT notation). Much of the sequential design has been reused where additional classes address parallel programming features. The plasma and field class objects are partitioned by partition region objects. A virtual parallel machine class object encapsulates features for portability.

slab-partitioning of the fields and particles is used. From an implementation point of view, using partition region objects as *members* of the field and plasma classes simplifies operations performed on the regions they own. For this reason, the `region` object is also included in the attribute list for the plasma and field classes.

The **ParticleTransportBuf** class provides a mechanism for moving groups of particles across processors since their position and velocity components are independent vector objects. The particle transport buffer class unifies (type converts) these objects into a single type so they can be transported together for efficiency. This reduces the number of particle message passing calls in half. Additional transport buffers are utilized, but they are defined within existing abstractions.

The **Plasma** class has new routines for partitioning—using a partition region object—and for moving particles across partition boundaries. The **Field** class also has an internal partition region class object to maintain information about the distributed field. Since the real force/charge density and complex Fourier fields share the identical partitioning in one dimension, these distinct fields can share the same partition region object. The makes the multiplicity of the link from the field to partition region class exactly one. Finally, a routine to move field guard cells across partition borders is included.

Table 2.3 on page 22 illustrates the major routines of the one-dimensional parallel Fortran 77 program. Reusing abstractions from existing codes simplified the interface to the major routines of the one-dimensional parallel C++ program in table 3.3. The routine parameters are straightforward and readable. The Poisson's Equation solver and the FFT are not included in table 3.3 since they are information hidden and not accessible to the main program. By including these two routines as members of the field class the computational grids are directly accessible.

The main program is shown in appendix A.8 on page 197. Although features for message passing have been included and some of the numerical routines have been modified to operate on distributed data, encapsulation incorporates these changes

Objects	`vpm, elec_pos, elec_vel, plasma, field, backgnd, beam, energy`
Call Syntax	`object.memberfunction()`
Definitions	The `vpm` object performs message passing, encapsulating parallel machine parameters. The `elec_pos` and `elec_vel` vector objects hold particle positions and velocities respectively. The `field` performs all field operations while `energy` stores the diagnostic.
Routine	`plasma.UniformSpcMaxwellVelDist(elec_pos, elec_vel, backgnd, vpm);` `plasma.UniformSpcMaxwellVelDist(elec_pos, elec_vel, beam, vpm);`
Action	**Initialize density profile and velocity distribution**. Distribute background and beam electrons using the plasma object.
Routine	`field.ChargeDeposition(elec_pos, plasma, ChargedParticle::e_charge);`
Action	**Deposit charge**. Deposit charge onto grid from background and beam electrons.
Routine	`plasma.Advance(elec_pos, elec_vel, field, energy, vpm);`
Action	**Push particles**. The plasma object advances particle positions and velocities using the field object while assigning the potential energy.
Routine	`plasma.Partition(vpm);` `field.Partition(vpm);`
Action	**Partition region**. Compute partitioning of plasma particles and field.
Routine	`plasma.UpdateDistribution(elec_pos, elec_vel, vpm);`
Action	**Particle mover**. Move particles to the processor that owns the spatial region.
Routine	`field.CalcEField( vpm, energy );`
Action	**Calculate Field**. Solve Poisson's Equation with FFT to convert charge density into electric force field. Also update guard cell data from particle partition to field partition and vice-versa.

TABLE 3.3: C++ major routines of the one-dimensional parallel program.

into the definition of the objects. In other words *object definitions have been extended for parallelism, but the interfaces have only been slightly modified.* Examining the Fortran 77 main program in appendix A.9 on page 199, there are major differences when compared to the sequential Fortran 77 and parallel C++ programs since abstractions cannot be represented as readily in Fortran 77. For instance, the parameters of the routines which distribute particles in the parallel Fortran 77 program:

```
c background electrons
      if (npx.gt.0) call pistr1 (part,edges,npp,nps,vtx,zero,npx,nx,
     1idimp,npmax,idps)
c beam electrons
      nps = npp + 1
      if (npxb.gt.0) call pistr1 (part,edges,npp,nps,vtdx,vdx,npxb,nx,
     iidimp,npmax,idps)
```

are less clearly described when compared to the same operations in the C++ parallel program:

```
plasma.UniformSpcMaxwellVelDist( elec_pos, elec_vel, backgnd, vpm );
plasma.UniformSpcMaxwellVelDist( elec_pos, elec_vel, beam, vpm );
```

The interface for higher dimensional programs remains unchanged in the C++ version, but the Fortran 77 version would become more complex as additional features are included.

Abstraction Modifications Introduced By Parallelism Although some of the particle operations were dimension-independent with the vector organization, this abstraction had to be reorganized into a new form (the particle transport buffer type) for interprocessor communication. Transporting particle positions and velocities separately—preserving the abstraction—is unreasonable since the communication overhead is higher than necessary. Apparently, some abstractions that are fine in a

sequential environment may cause performance penalties in a parallel environment. Our temporary solution was to change the form of the abstraction to facilitate parallel computation concerns, but was this reasonable? For object-oriented techniques to be useful in scalable parallel computing existing *correct* abstractions should be preserved whenever possible. Approaches that consider these issues must be part of the overall design process. We may proceed by allowing modification of some abstractions to preserve consistency with previous work. Alternatively, new approaches can be investigated to introduce consistency for parallelism that may render previous approaches at least partially obsolete. These options will be studied as more advanced simulation programs are designed.

3.3.2 Two-Dimensional PIC Simulation

Modifying the definition of particles, fields, and the numerical solvers is required to extend the one-dimensional code into two dimensions. Encapsulating abstractions affecting the dimensionality of the problem streamlines program extension. The definition of the GridPoint<T> template class simplifies addition of the y force component to the entire field. The PointVector2D<T> template class, inherited into the charged particle class, creates a two-dimensional vector space of charged particles from the Vector<T> class object. All of these interface modifications are straightforward and immediate from the object-oriented hierarchy presented in figure 3.6 on page 48.

These modifications allow the main program of the two-dimensional parallel version to have exactly the same interface as the one-dimensional parallel program. Nevertheless, the design still requires changing the abstraction model for particles when interprocessor communication is performed. Since the vector space distribution of particles must necessarily require more parameters in two dimensions, the initialization of the background and beam particle distribution objects are affected. The object types are identical so this does not affect the definition of the interfaces, as seen in appendix A.8 on page 197.

3.3.3 Three-Dimensional PIC Simulation

The design of the two-dimensional parallel code using the vector space abstractions was very successful, so it makes sense to reuse these concepts for the three-dimensional program. In figure 3.7 we show the class hierarchy which is an extension of the one-dimensional version from figure 3.6 on page 48. The structure of the hierarchy for two-dimensions is exactly the same, where the z components would be omitted. One interesting difference from the one-dimensional version involves the partitioning of the three-dimensional field. The one-dimensional parallel program uses a fully parallel Fast Fourier Transform algorithm, so only one partitioning of the field is required—the charge density and force fields may share the same partitioning. The higher-dimensional programs perform Fast Fourier Transforms using a sequential algorithm with matrix transposes for data distribution across processors. This requires real, complex, and transposed complex partitioning managed by three partition region objects indicated in figure 3.7.

The interface of the main program matches that of the lower dimensional versions seen in appendix A.8 on page 197. There are issues of interest, however, beyond the surface of the main program. The field solver operations were given direct access to the field components which simplifies program development in C++. Unfortunately, this may not be the abstraction we desire. If the field component names change then the FFT and Poisson's Equation solver must be modified since they use these names directly. Also, if machine specific vendor optimized Fourier transforms were available, these routines would not be designed with our class naming convention in mind. Our interface to the Fourier transform could remain, but we might want to change the form of the field abstraction to use the optimized vendor routines, converting back to our abstraction later. The vector organization provided flexibility and ease in the transition to higher dimensional codes through abstraction of dimension dependent operations. *This was beneficial, but a disappointing realization is that full vector operations have seen limited usage in the program. The advantageous feature of this*

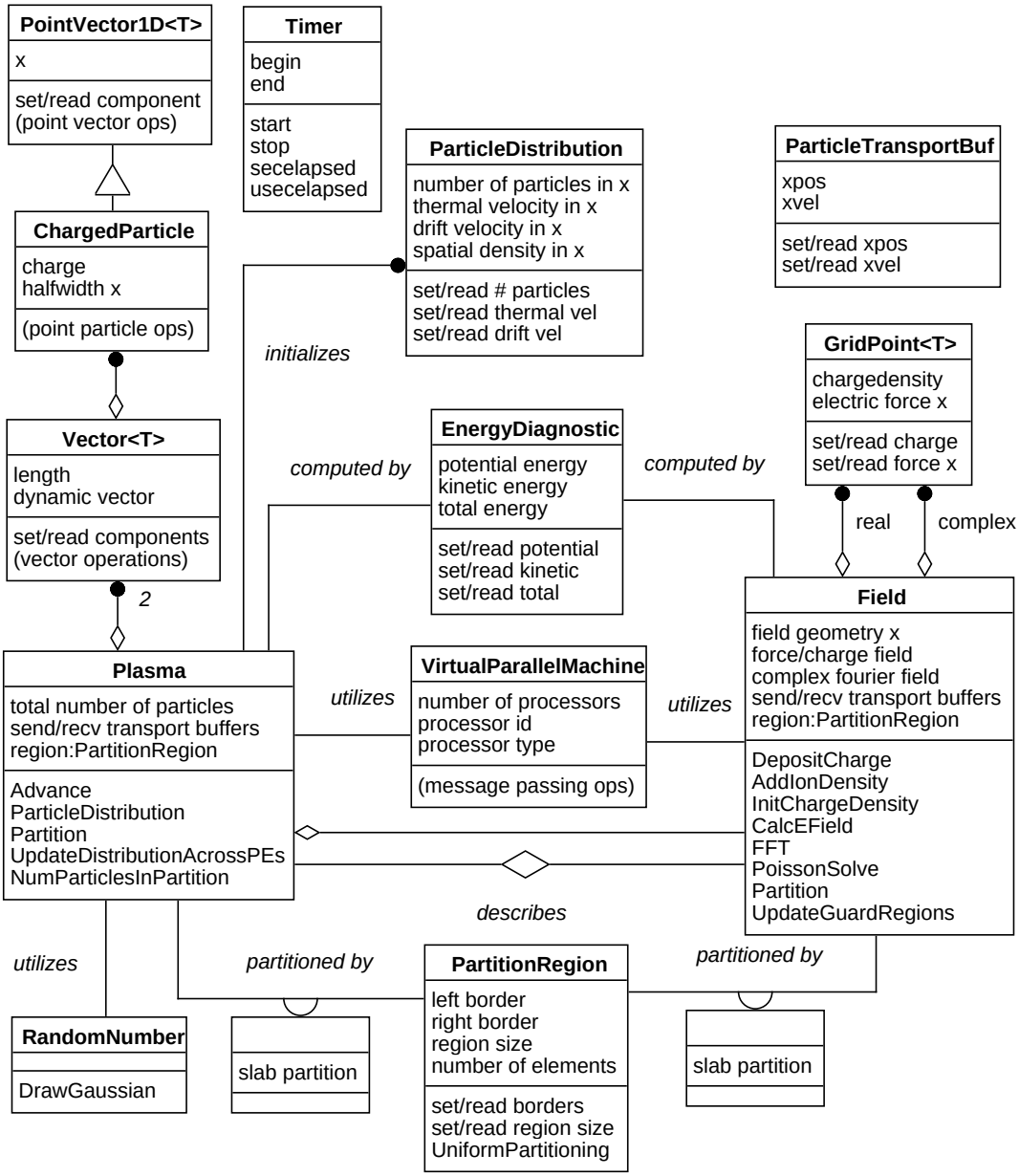


FIGURE 3.7: C++ three-dimensional parallel class hierarchy (OMT notation). The abstractions from lower-dimensional programs are preserved. The template structure and encapsulated features simplify extending objects into three dimensions.

abstraction really involved the encapsulation of dimension-dependent concepts, not the mathematical vector operations.

Rethinking The Design For Advanced Programming When thinking about the design of future codes, dynamic memory management of dimension-dependent fields would be useful for load balancing. Also, from a physical point of view, the charge density field is a *scalar field* (single component) while the electric force field is a *vector field* (multiple components). The ability to create scalar and vector fields of various types would be useful now, and for dynamic load balancing in the future. Our experience with building abstractions indicates that such a model can clearly be created.

In figure 3.8 we show a modern three-dimensional parallel code design with a variety of useful features. Although this design may look very different, it shares much in common with the previous versions. The **Species<T>** template class creates various particle species which are automatically partitioned across processors. Using the `Particle3D` type, a species of electrons partitioned by the **SlabParition** class in three dimensions can be created. Since the link association between the species and the partitioning is defined as slabs from the slab partition class, the OMT link association symbol has been removed from the diagram. The kinetic energy of the system is associated with the particle species, so this attribute is now part of the species class. The species can be distributed based on the particle distribution class objects that describe properties of the background and beam electron species.

The field organization has changed dramatically. The **ScalarField3D<T>** class is used to create the charge density field, while a **VectorField3D<T>** represents the electric force field. Once again, a **FieldPoint3D<T>** class parameterizes the computational grid point elements for vector fields. The scalar and vector fields are derived from a **GeneralField3D** class which provides geometry and partitioning information. These fields can be dynamically resized, which will be useful as fields are

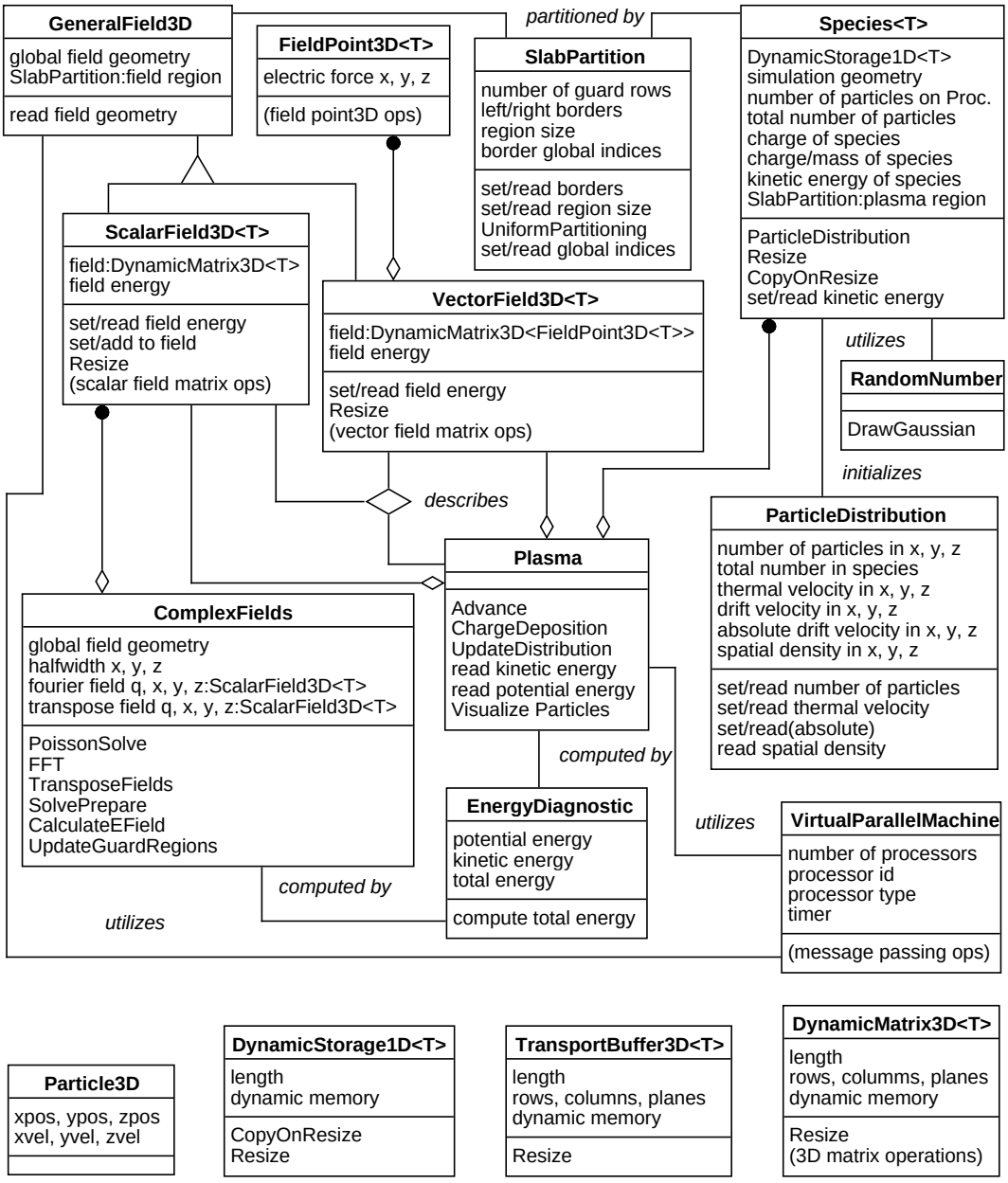


FIGURE 3.8: Alternative C++ three-dimensional parallel class hierarchy (OMT notation). Scalar and vector fields have been introduced, as well as complex fields used by numerical field solvers. A parameterized species class replaced the vector organization of particles, maintaining the abstraction during interprocessor communication. The field and particle partitioning can be dynamically resized automatically.

modified due to load balancing. The Fast Fourier Transform and Poisson's Equation are solved on complex fields, which differ in type and geometry from the scalar and vector fields. Since these fields also have transposed components and are only used for computation, they have been separated from the other fields. The **ComplexFields** class supports these operations where the particle halfwidth components—used by the Poisson's Equation solver—have been incorporated. Scalar fields are used in constructing the complex fields.

The plasma class performs the major simulation operations since this abstraction consists of the scalar charge density field, vector electric field, and the electron particle species. Although energy diagnostics are computed elsewhere, this class is responsible for reporting diagnostics maintained by the energy diagnostic class. The ternary association among the scalar field, vector field, and plasma indicates that fields can describe particles and vice-versa. Some of the additional classes used in the construction of other classes are shown included in the diagram. In particular, the **TransportBuffer<T>** classes are useful in transmitting field and particle species across processors. Also, the **DynamicStorage<T>** and **DynamicMatrix<T>** classes are useful in constructing fields.

Appendix A.10 on page 201 shows the interface for the main program. In this organization charge density and electric fields are separate, allowing operations to be tailored to the fields required. In the previous programs, where charge density and electric field components were encapsulated into one structure, operations requiring only one type of field had access to both. Since charge deposition does not require the electric field, and since advancing particles does not require the charge density field, these operations now appear as follows:

```
plasma.ChargeDeposition( electrons, cdensity );
plasma.Advance( electrons, efield, DT );
```

where `DT` is the time step. This model will be used in the development of the more advanced simulation codes.

3.3.4 Commentary on Parallel Models

We are now prepared to address some important issues that have intentionally been omitted. These issues involve object management in scientific parallel programs.

Modeling Concurrency with Objects In our approach objects view themselves as communicating sequential processes that possibly represent and/or operate on distributed data. Some objects, such as the particles, simply move among processors unaware that they are participating in a parallel computation since their *definition* includes no information about a distributed environment. Nevertheless, since particles are part of a distributed species—by definition of the species class—they participate in the parallel computation. Other objects, including the plasma, charge density, and electric fields, are also aware of the distributed computation since partitioning information is encapsulated into their class definition. For example, when the charge density field is created with a global geometry everything required to define this partitioned object is handled *automatically* by the slab partitioning object bound to the scalar field class definition. In other words, *the parallelism arises from operations performed on objects that are distributed by definition.* Objects that are not distributed are handled normally, or may become part of a distributed computation once bound to an object which is distributed. This approach is reasonable for SPMD object-oriented programs where each processor executes the same program, but on distributed data.

Nonetheless, alternative approaches for representing parallelism for plasma computations exist—although they may not necessarily take object-oriented aspects into consideration [17]. Parallelism can be introduced by creating a single object on one node which spawns multiple copies of itself distributed across the processors. Specialized annotations can be added for concurrency and interprocessor communication which may include data parallel language extensions, software supported shared memory programming, or different parallel programming models including the Bulk

Synchronous Parallel (BSP) model [40].

Since this plasma code is generally data parallel and not task parallel, encapsulating aspects of parallelism into the definition of objects was the best approach for our needs. It was more direct to control global addressing, the namespace, and ownership of regions by representing parallelism through the creation of objects that encapsulated this information in their definitions (implemented by the owner-computes rule), rather than applying an alternative approach. Objects are specified using global dimensions and properties, but the parallelism, data management, operations, and storge management are handled by the automatic partitioning of objects on creation. This approach allows for simple and direct extension of a non-distributed data object into a distributed data object useful in parallel programming. Table 3.4 shows the main routines of the modified three-dimensional C++ parallel program.

3.4 Evaluation, Discussion, and Advanced Issues

As sequential and parallel models were proposed and implemented abstractions were continually modified. This experience allows evaluation of the abstraction process in scientific programming with discussion of general issues applicable to these kind of problems.

Evolving Basic Programs to Advanced Versions The original Fortran 77 codes could not allow for object-oriented abstraction modeling due to limitations in language features. As abstraction features were added to the C++ programs such as particle species, parameterized computational fields, classes for component interaction, and features for parallelism including partitioning and object based message passing, a specific model always guided the design. Our model involved decomposing particles and fields into independent regions distributed across processors. This was a natural extension of the sequential design that simplified extending these abstractions into the advanced parallel codes. This allowed particle and field partitioning to be han-

Routine	`electrons.UniformDistribution( backdf, vpm );` `electrons.UniformDistribution( beamdf, vpm );`
Action	**Initialize density profile and velocity distribution**. Distribute background and beam electron species using the distribution objects.
Routine	`plasma.ChargeDeposition( electrons, cdensity );`
Action	**Deposit charge**. Electron species deposits to charge density field.
Routine	`plasma.Advance( electrons, efield, DT );`
Action	**Push particles**. Advances species positions and velocities using the electric field while assigning field energy.
Routine	`plasma.UpdateDistribution( electrons, vpm );`
Action	**Particle mover**. Move particles to the processor that owns the spatial region.
Routine	`fields.Solve( cdensity, efield, vpm );`
Action	**Calculate Field**. Solve field equations using charge density and electric field on distributed data.
Routine	`plasma.pe( efield, energy );` `plasma.ke( electrons, energy );`
Action	**Energy**. Find the energy of the field and particles.

TABLE 3.4: C++ major routines of the three-dimensional parallel program. Concurrency features are encapsulated in object definitions. Additionally, abstractions have been modified to represent scalar and vector fields independently.

dled separately where the interactions among abstractions was clearly defined. For instance, the particle species could participate in operations involving the charge density field or the electric field. These fields could also be converted to complex fields for the solution of Maxwell's Equations without interfering with any other abstraction. *This "separation of concerns" with facilities for interaction simplifies the design and extension of basic codes to more advanced codes.* The benefits of this approach will be seen in Chapter 5.

Other models have been proposed that support abstraction, but extension may

not be as immediate. One model binds portions of the grid directly to each particle making the near-neighbor grid points immediately accessible.[5] It is unclear that this approach is worth the effort involved in unifying these grids to solve the complete field. Furthermore, defining a particle to include a portion of the grid may not be a proper abstraction in the first place. If particles include field components in their definition then particles would be involved in field solvers which is not appropriate. *The creation of abstractions in scientific computing must provide a mechanism for adding new features without affecting existing abstractions.* For example, our designs allow an ion species to be added without affecting the definition of the fields or the field solvers.

Inheritance and Templates Inheritance, while sometimes appropriate, had limited usage in the design of our class hierarchies. Most general purpose object-oriented textbooks state that inheritance is an essential part of code reuse and application programming. In contrast, *our designs actually removed inheritance-based definitions of various particle types since template parameterization of object construction provided greater flexibility.* The physical description of electrons and ions sharing an IS-A relationship with a base class particle type may seem reasonable. Nevertheless, if distinguishing characteristics of computational objects (such as the charge on a particle) can be parameterized during object creation rather than through inheritance, then perhaps this notion is not that critical in scientific programming.

Our experience in writing many models—beyond what has been presented—indicates that *inheritance should only be used in scientific programming when there is more to gain than the minor extension of an existing concept.* Inheritance hierarchies are difficult to maintain, particularly when sub-typing inheritance is used. In scientific abstraction modeling if a derived class object does not preserve the exact semantics of the base class object the meaning of the IS-A sub-typing relationship is destroyed.

[5]Personal communication during the poster session of the 15th International Conference on the Numerical Simulation of Plasmas. King of Prussia, Pennsylvania. September, 1994.

There are not many instances in our experience where preserving sub-typing was useful, rather the ability to compose abstractions in a HAS-A relationship was more important. One example was the incorporation of partition objects encapsulated into existing abstractions allowing for distribution across the parallel processors.

When most of the extensions to a program consist of changing the properties of an abstraction rather than building new abstractions, the use of well-designed templates simplifies code modifications and the introduction of new features. Returning to particle modeling, in this case a new abstraction using inheritance is not always necessary to distinguish an electron from an ion; modifying the properties of the general species definition through parameterization is easier and more direct. There are, however, times when creating a new abstraction may be useful—ions may require a different particle advance routine. Distinguishing objects based on type, preserving type safety, is an important feature that inheritance mechanisms can provide which found important use in our field definitions. If objects are only distinguished by parameterized features, however, then inheritance may not need to be introduced.

There are various forms of inheritance that exist [41] and often a programmer's definition of inheritance is based on the language that is most familiar. In abstraction modeling for scientific computing our designs indicate that sub-typing inheritance (type-preserving inheritance where the derived class shares every property of the base class) has not been as useful as composition inheritance (aggregation-inheritance where classes are composed of features from existing classes, but subtypes are not necessarily created). Discussions of the role sub-typing plays in the definition of inheritance is not new [6, 13]. Often, however, programmers may consider this issue the defining subject of how well a language supports object-oriented programming—including C++ since sub-typing is automatically provided in that language. Scientific applications have different requirements than other applications where the object-oriented methodology is applied, and our impression is that this paradigm will be applied differently in this domain. This question will be raised again in chapter 4 as

the ways in which Fortran 90 supports object-oriented programming is explored.

Relevance To Fusion Code Modeling Examining the usefulness of encapsulation, inheritance, and other object-oriented features has been identified as an important component toward the development of modern fusion codes. Many of these features have allowed the internal numerical and structural organization of the PIC application to be modified while preserving well defined interfaces and computational abstractions. The ability to view some abstractions as individual components and as a collection was important, for example, in particle modeling. Additionally, manipulation of abstractions with support for their interaction influenced the various field models, as well as the particle species and field interactions. Future chapters will consider the implications of abstraction modeling on programming design decisions, language selection, extension to new simulation problems, and the performance effects of programming in the C++ and Fortran 90 languages.

Chapter 4

Object-Oriented Programming in Fortran 90

This chapter introduces how object-oriented concepts are supported by the Fortran 90 programming language. The principal features of object-oriented methodology are described with examples from our sequential and parallel PIC programs. Only the general issues are presented. For more in-depth discussions see [8, 9, 45, 46].

4.1 Fortran 90: The New Standard

The C++ programming language [36, 60] is well-known for its support of object-oriented concepts useful in abstraction modeling. Containing many important features, its popularity is growing with a new generation of scientists anxious to bring clarity and flexibility to their programming efforts. Nevertheless, most of the scientific applications in development and use today are based on Fortran—the most popular language for scientific programming.

Fortran is not a static language; it has continually evolved, somewhat slowly, to include the most recent proven ideas and concepts garnered from other programming languages. Until recently, modern aspects of software design were not supported

which complicated abstraction modeling for large scale development projects. This can make software difficult to comprehend, unsafe, and potentially problematic. The emergence of Fortran 90 [11] has dramatically altered traditional Fortran programming. Many modern programming language techniques are included in the standard with new features that will influence practical scientific programming [46]. We believe these features extend far beyond the well-publicized array-syntax operations, into the object-oriented programming paradigm.

Although Fortran 77 programs are completely compatible with the Fortran 90 standard, Fortran 90 is a very different language. The new constructs encourage the creation of abstract data types, encapsulation, information hiding, inheritance, generic programming and many features to ensure the safe development of advanced programs. These new features have ushered in Fortran 90 as a modern language whose benefits can be compared and evaluated with other modern languages, including C++. Our goal is to illustrate how Fortran 90 supports object-oriented programming concepts, not to teach this language. An excellent textbook is "Fortran 90 Programming" by Ellis, Philips and Lahey [11]. Sometimes, Fortran 90 terminology matches that of other languages, yet the meaning differs. We will try to distinguish these instances as clearly as possible.

Finally, the features of Fortran 90 are quite powerful by themselves, even if the object paradigm is not applied. For experienced Fortran 77 users looking to modernize their programs to enhance collaboration, usability, sharing, and software extension, one benefit is that the new aspects of Fortran 90 can be incrementally introduced into existing Fortran 77 programs. This allows programs to evolve in the context of a well-known environment, permitting scientific productivity to continue while new ideas are acquired. Unfortunately, there is always some risk associated with change—using Fortran 90 most effectively requires a paradigm-shift. Some reluctance is natural given the investments in existing software and process of code development. Nevertheless Fortran 90 is a standard, which when applied properly, can make scientific program-

ming very clear and productive, particularly when object-oriented methodology is applied. We will explore these ideas throughout this chapter and perform a detailed comparative analysis to Fortran 77 and C++ in Chapter 6.

4.2 Object-Oriented Fortran 90 Programming

The now familiar routines of advancing particles and computing their charge deposition from the original Fortran 77 one-dimensional program are:

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

where an array of particles (`part`), charge density field (`q`), and electric force field (`fx`) are used in the operations of pushing particles (`push1`) and depositing charge (`dpost1`). Additional parameters include the charge on an electron (`qme`), the time step (`dt`), field dimension (`nx`), particle kinetic energy (`wke`), and number of particles in the electron species (`np`). Even in this example, the code would be clearer and easier to manipulate if particle and field information were *encapsulated* into logically related units:

```
USE plasma_module
TYPE (species) :: electrons, ions
TYPE (fields) :: charge_density, efield
real :: dt = .2
call plasma_push1 (electrons, efield, dt)
call plasma_dpost1 (electrons, charge_density)
```

In this Fortran 90 example the collection of electrons, with their properties, are localized to the definition of the `species` type. Additional species can be created, including ions, due to the encapsulation of species features. Similarly, the properties of fields are encapsulated into a `fields` type from which the charge density and electric field are created. The details of how these fields are stored and manipulated

are hidden in the definition of the type. This notion, related to encapsulation and known as *information hiding* supports creation of user defined types called *abstract data types*. Together, encapsulation, information hiding, inheritance, and abstract data types form the building blocks of object-oriented programs.

Comparing the `plasma_push1` and `plasma_dpost1` calls to the `push1` and `dpost1` calls, the parameters have been simplified representing problem-related abstractions. These new data types and associated operations can be combined into a *module* for coherency. Modules can be used in parts of the program that require access to the routines and data they provide. For added protection, the internal details of the module can be hidden (*private*). Then, the only way to manipulate module data is with the functions and subroutines the module makes publicly accessible. The private data will always be directly accessible to the routines defined within the module. This provides a clean interface adding clarity, protection, and a stable context for information sharing and collaboration useful for increasingly complex scientific programs.

In the following sections we introduce many, but not all, of the new Fortran 90 constructs that can support object-oriented programming. The terminology will be defined and introduced with small examples from various programming segments. Due to space considerations, we may omit detailed language statements within our general discussion. See, for example, [11] for formal descriptions of the language statements. In later sections, we illustrate full scale programs with comparison to the object-oriented models of Chapter 3. Although some of the Fortran 90 programs were developed and modified simultaneously with the C++ programs, for clarity, we present them independently in this chapter.

4.2.1 Encapsulation with Derived Types

Fortran provides a number of intrinsic data types including integer, real, complex, character, and logical. In object-oriented programming, it is useful to create user-

defined types from intrinsics and/or previously defined new user types. Fortran 90 allows this through the creation of *derived types* which allow for encapsulation of related concepts into a single form. A derived type can be created to describe the distribution properties of particles in Fortran 90:

```
TYPE distribution_function1d
   INTEGER :: number_of_particles_x
   REAL :: thermal_velocity_x, drift_velocity_x, spatial_density
END TYPE distribution_function1d
```

where the type has the name `distribution_function1d` with components describing the information required for particle distribution. Variables of this type can now be created using the syntax:

```
TYPE (distribution_function1d) :: backdf, beamdf
```

and passed as parameters to various procedures (if defined in a module or through an explicit interface) since derived types have many of the capabilities of intrinsic data types. The percent symbol is used to access components of derived types. Specifying the number of particles and drift velocity of the background distribution is straightforward:

```
backdf%number_of_particles = 20000
backdf%drift_velocity_x = 5.0
```

Derived types can use previously defined derived types in their definition.

In object-oriented programming we often create user defined types with operations bound to the type. This provides an interface to modify the components of the type where the internal details are hidden. Our current use of Fortran 90 derived types does not provide a mechanism for defining a set of operations on the type. How, for example, should we create a distribution? The best way is to define the derived type within a Fortran 90 *module*, seen in figure 4.1. The number of particles distributed (`npxs`), the system length (`nxs`), the thermal (`vtxs`) and drift (`vdxs`) velocity of the

Sketch of Fortran 90 module with a derived type

```
MODULE distribution_module
IMPLICIT NONE
TYPE distribution_function1d
   INTEGER :: number_of_particles_x
   REAL :: thermal_velocity_x, drift_velocity_x, spatial_density
END TYPE distribution_function1d
CONTAINS
   SUBROUTINE distribution_create(distf,npxs,nxs,vtxs,vdxs)
   TYPE (distribution_function1d), INTENT (out) :: distf
   INTEGER, INTENT (in) :: npxs, nxs
   REAL, INTENT (in) :: vtxs, vdxs
      distf%number_of_particles_x = npxs
      distf%thermal_velocity_x = vtxs
      distf%drift_velocity_x = vdxs
      distf%spatial_density = float(nxs)/float(npxs)
   END SUBROUTINE distribution_create
END MODULE distribution_module
```

FIGURE 4.1: Sketch of a Fortran 90 distribution function module that contains a routine to initialize distribution derived type objects.

particles, and the distribution characteristic (`distf`) are formal parameters to the `distribution_create` function.

By defining a module an explicit procedure interface for the distribution routine has been created. Features of the module can be used in various routines, and other modules, as needed. The `distribution_create` routine takes the derived type as its first argument, followed by the features of the distribution that characterize the type variable. Notice that the formal parameters are fully typed. The `intent(in)` attribute indicates which formal parameters cannot be modified within the routine; they are for input only. The `intent(out)` attribute means that the argument is undefined on entry to the routine where it must be assigned a value before evaluation in any expression; it is for output only. Bidirectional attributes can be specified with `intent(inout)`. The `contains` statement, which is part of the module, indicates that a list of routines the module makes available follows. One of the benefits of including

routines within modules via the contains statement is that an explicit interface is created, so argument type checking is possible. For example, if the actual arguments to a function or subroutine call do not match the formal parameter interface, a compile time error will occur. Finally, the `implicit none` statement prevents Fortran 77-style implicit variable creation—all variables must be declared before they are used.

Initializing the background and beam electron distribution functions is straightforward:

```
TYPE (distribution_function1d) :: backdf, beamdf    ! create
call distribution_create(backdf,npx,nx,vtx,zero)    ! initialize
call distribution_create(beamdf,npxb,nx,vtdx,vdx)
```

provided the module is visible to the program calling unit. Variables created from derived types defined within a module are the *objects* of Fortran 90 object-oriented programming.

4.2.2 Overview of Modules

The module is a critically important new addition to Fortran 90 since modules increase the visibility and accessibility of data and routines throughout the program. Modules are much more powerful than this however, they can easily be used to support the object-oriented methodology. Modules allow encapsulation of derived types and the routines that operate with them; they can be "used" wherever module variables (objects) are needed. The *use* statement allows the features of modules to be accessible to any program unit. For example, we can create a module for the particle species which requires features of the distribution function, seen in figure 4.2.

Through *use-association*, access to the `distribution_module` derived types and features is now provided to the `species_module`. A function to distribute the species, given the distribution properties from the `distf` object, can be defined in the species module after the `contains` statement:

```
SUBROUTINE species_distribute (species,distf)
```

Sketch of Fortran 90 module with use-association

```
MODULE species_module
USE distribution_module     ! access additional features
IMPLICIT NONE
TYPE particle1d
   REAL :: x, vx            ! x position and velocity
END TYPE particle1d
TYPE species
   REAL :: qm, qbm, ek      ! charge, charge/mass, energy
   INTEGER :: nop           ! number of particles
   TYPE (particle1d), DIMENSION (:), POINTER :: p
END TYPE species
CONTAINS
   SUBROUTINE species_distribute (species,distf)
   ! details omitted...
   END SUBROUTINE species_distribute
! list of additional routines...
END MODULE species_module
```

FIGURE 4.2: Sketch of a Fortran 90 collective species module that uses a distribution function module for spatial and velocity distribution of various particle species.

```
TYPE (species1d), INTENT (out) :: species
TYPE (distribution_function1d), INTENT (in) :: distf
   ! uniform density profile
   do  j = 1, distf%number_of_particles_x
       species%p(j)%x = distf%spatial_density*(float(j)-.5)
   enddo
   ! remainder of routine...
END SUBROUTINE species_distribute
```

The `distf` object has direct access to its components through use-association of the distribution module. Note that Fortran 90 loops do not require labels and that free-format source code programming is supported.

Returning to the species module, notice that in addition to the distribution features a `particle1d` type and a `species` type have been added. The species includes the charge (`qm`), charge/mass (`qbm`), species kinetic energy (`ek`), and the number of

particles in the species (`nop`) as part of the encapsulated definition. A pointer to a one-dimensional array of particles is also included in the species derived type. Fortran 90 supports pointer types and dynamic memory allocation, however, we should explain why this pointer structure was introduced. Derived types have some restrictions on their content. We require storage for an array of particles as part of the species derived type definition. If the number of particles is known at compile time, the derived type could contain the statement:

```
TYPE (particle1d), DIMENSION (NumberOfParticles) :: p
```

However, the number of particles may be determined dynamically implying that storage must be allocated at runtime. An *allocatable array* in Fortran 90 allows the programmer to create and destroy arrays dynamically, *but allocatable arrays are not allowed in derived type definitions.* Pointers to arrays are allowed, explaining the use of the pointer attribute:

```
TYPE (particle1d), DIMENSION (:), POINTER :: p
```

where a pointer to a one-dimensional dynamically allocated array is declared. Storage allocation occurs in a separate executable statement.

Use-association gives the `species_distribute` routine direct access to the distribution function derived type, since the components of the `distribution_module` were public. However, we may want to restrict access for protection or information hiding. Any component of a module can be made private to the module, limiting external access to its internal management while creating a standard interface to the features the module provides. Indeed, the designer has complete control over the accessibility of any component of the module. This includes data, derived types, and routines which may be specified as *public* or *private* explicitly. When a derived type is in a module, its components may be private while the type itself is still publicly available wherever the module is used. If the following derived type were defined in the `species_module`

```
TYPE particle1d
   PRIVATE
   REAL :: x, vx       ! x position and velocity
END TYPE particle1d
```

where the position and velocity are declared, then objects of type `particle1d` can be created by use-association of the module, but the components are only accessible to routines defined within the module or by an interface that the module must provide. Alternatively the derived type may be entirely private to the `species_module`:

```
TYPE, PRIVATE :: hidden_particle1d
   REAL :: x, vx
END TYPE hidden_particle1d
```

Now, variables of type `hidden_particle1d` cannot be created outside of the module by use-association; the type is hidden. These examples illustrate two forms of information hiding.

All modules have a default accessibility of public unless explicitly specified otherwise. The accessibility to each component of a module can be listed explicitly, or overridden, if desired. To encourage information hiding in object-oriented programming the default accessibility of the module can be private, followed by a list of routines and data that are publicly accessible by the `public` attribute. Modules should be used as much as possible in object-oriented Fortran 90 programming since they provide the only mechanism to gain access to the advanced features of the language. They will have an important role in the construction of inheritance relationships in Fortran 90.

4.2.3 Inheritance and Related Issues

Inheritance, the ability to create new abstractions from existing abstractions, is fundamental to object-oriented programming. Fortran 90 can support inheritance with the *use* statement, since one module (the derived) can gain access to another module

(the base) by use-association. Object-oriented languages define inheritance in various ways—often the separate issues of sub-typing and reuse inheritance are unified in some languages leading to misconceptions [13, 41]. In Fortran 90, inheritance allows modules with sophisticated derived types to be created from modules with simple derived types, since objects are created from derived types in modules. Similarly in C++, inheritance allows objects created from complicated classes to be formed from simple classes. There is a subtle distinction in these statements that will be addressed shortly.

The simplest form of inheritance in Fortran 90 is to provide access to base module data by use-association into a derived module:

```
MODULE base_module
   IMPLICIT NONE
   INTEGER :: base_data
END MODULE base_module

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

The module `derived_module` can now be used in a program segment where both `base_data` and `derived_data` will be accessible:

```
PROGRAM inherit_test
   USE derived_module
   IMPLICIT NONE
   ! this program can access base_data and derived_data...
END PROGRAM inherit_test
```

It would be more useful, however, to create new derived types from derived types defined in existing modules. Additionally, we want to create routines that know what action to perform, automatically, based on the type of the object when inheritance is

used. It would also be helpful if the derived module routines could call base module routines in their definition.

Inheritance By Composition The most general definition of inheritance allows complex classes to be composed from simple component classes. As an implementation technique, *inheritance by composition* allows new classes to be defined as incremental changes to existing classes [14]. For example, given a base module that contains a routine to advance a particle, called `push`, we may want to create a derived module for advancing a collection of particles. Additionally, `push` should be callable by a base module or derived module object where the action performed depends on the object type:

```
MODULE base_module
IMPLICIT NONE
! particle derived type definition...
INTERFACE push
   MODULE PROCEDURE push_one_particle
END INTERFACE
CONTAINS
   SUBROUTINE push_one_particle(this)
   TYPE (particle), INTENT(inout) :: this
      ! push operation...
   END SUBROUTINE push_one_particle
END MODULE base_module
```

When `push` is called by a base module object, the `interface` statement calls the `push_one_particle` routine provided the argument (`this`) is of type `particle`. Similarly, the derived module defines `push` as an interface to the `push_all_particles` routine. In other words, we have *overloaded* the routine `push`, and the proper routine will be called based on the type of the object. The derived class is illustrated in figure 4.3. Since the derived module has "used" (or inherited) the base module, any program unit that uses the derived module/class will also have access to the public part of the base module/class. This allows the derived class to call the `push` routine,

Sketch of Fortran 90 Inheritance by Composition

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

FIGURE 4.3: Sketch of composition inheritance in Fortran 90 where the derived class uses base class operations in implementing member routines.

defined in the base class, in a loop to push a collection of particles.

Fortran 90 arrays are actually objects, therefore, the `size` intrinsic will return information about the size of the array. Notice that the loop index variable `i` was *not initialized* on declaration. If we initialized this variable it would be static; the final value of `i` would be retained on subsequent calls to the routine. In C++, it is common to initialize loop variables on declaration; *remain aware that careless initialization of variables in Fortran 90 may cause unexpected results.*

When the derived module is used in any program unit and `push` is called, either the base or derived module version will execute depending on the type of the object, as seen in figure 4.4. Notice that the `parameter` attribute was used to define the size of the array (`nx`) as a constant.

We have seen how the `use` statement allows inheritance by composition of data and routines where interfaces may be created allowing base and derived modules to

Operations on Objects Defined by Composition

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

FIGURE 4.4: Usage of Fortran 90 objects defined through composition inheritance. The construction allows a single overloaded operator to push a single particle, or a group of particles, where the derived class function uses the base class operations in its definition.

share routine names. This sharing is permitted because the actions performed are controlled by overloading the routines based on the derived types. However, can inheritance in Fortran 90 be defined so that new derived types can be created from existing derived types? This form of sub-typing inheritance, which depends on the typing features of specific programming languages, is studied in the next section.

Inheritance By Sub-Typing Inheritance in the C++ programming language allows a derived class object to assume the base class type through an implicit type-conversion, if necessary. This implies that derived class objects can be used wherever base class objects are expected, because the derived object is a base object; this is called an IS-A relationship. This is possible since a derived object contains *all* the components of its class and every ancestor class—*C++ objects are created from classes of a specific type.* Fortran 90 modules consist of a collection of derived types with associated operations involving those types, where objects are created by use-association of the module. In other words, *Fortran 90 objects are created from derived types via modules.* We hinted earlier that there was a distinction between C++ classes and

Fortran 90 modules that influence how inheritance is modeled. We are now prepared to explore the implications of this distinction.

Inheritance of a base class in C++ modifies the structure of the derived class. Base class attributes are physically added to the derived class. This is why C++ derived class objects also "inherit" the type of the base class; it allows the base class methods, which have been added, to be called on derived class objects. Since Fortran 90 objects are created from derived types (not from modules which only make derived types available) and since the `use` statement does not modify the structure of the derived module, inheritance in Fortran 90 means that we get functionality, not typing features. The principal reason for this is that *the Fortran 90 typing system does not allow for implicit type conversions.* Rather, Fortran 90 objects created from the derived module have access to the base module routines and derived types by use-association, but they have not been modified by the base module derived type's components.

Nevertheless, it is still possible to construct derived type objects in the derived module from base module derived types via inheritance, but we must be explicit. In C++, the compiler implicitly (automatically) includes the base class components in the derived class object definition. Interestingly enough, the explicit Fortran 90 approach is very powerful since base module derived type information can be combined into derived module derived types in non-trivial ways. We will see in later sections the important implication of this scheme in abstraction modeling for PIC simulation.

As another purely academic example, we can create an electron derived type from a particle derived type using inheritance, where we have included a function which will print the position of a particle. Using inheritance, we can extend the definition of `particle` objects into `electron` objects, as seen in figure 4.5.

Since the `particle` derived type has publicly accessible components, we could have accessed them directly from within the `e_pos` routine:

```
SUBROUTINE e_pos(this)
TYPE (electron), INTENT(in) :: this
```

Sketch of Fortran 90 Inheritance by Sub-Typing

```
MODULE particle_module
IMPLICIT NONE
TYPE particle
   REAL :: x, vx
END TYPE particle
INTERFACE position
   MODULE PROCEDURE pos
END INTERFACE
CONTAINS
   SUBROUTINE pos(this)
   TYPE (particle), INTENT(in) :: this
      print *, this%x
   END SUBROUTINE pos
END MODULE particle_module

MODULE electron_module
USE particle_module
IMPLICIT NONE
TYPE electron
   REAL :: charge
   TYPE (particle) :: p
END TYPE electron
INTERFACE position
   MODULE PROCEDURE e_pos
END INTERFACE
CONTAINS
   SUBROUTINE e_pos(this)
   TYPE (electron), INTENT(in) :: this
      call position(this%p)       ! access base class routine
   END SUBROUTINE e_pos
   SUBROUTINE e_charge(this)
   TYPE (electron), INTENT(in) :: this
      print *, this%charge        ! access electron component
   END SUBROUTINE e_charge
END MODULE electron_module
```

FIGURE 4.5: Usage of Fortran 90 objects defined through sub-typing inheritance. The construction allows a single overloaded operator to access features of objects defined through sub-typing inheritance.

```
      print *, this%p%x          ! access particle component
   END SUBROUTINE e_pos
```

However, modifying base class internal data through the public interface provided is often safer as programs are modified and reused.

When the electron module is used in any program unit `particle` and `electron` objects can be created, where the electron has been defined through inheritance of the particle derived type. Additionally, base class routines have been extended to work in the derived class—note that the base class was not modified in this construction. This illustrates *inheritance by sub-typing* since derived objects can use base class routines as if they were the same type, by overriding existing definitions, as illustrated in the example below:

```
PROGRAM inherit_test
   USE electron_module
   IMPLICIT NONE
   TYPE (particle) :: p
   TYPE (electron) :: e
   ! initialization code omitted...
   call position(p)              ! print position of particle
   call position(e)              ! print position of electron
   call e_charge(p)              ! ERROR, function not defined
   call e_charge(e)              ! print charge on electron
   stop
END PROGRAM inherit_test
```

Note that this is a construction; no automatic type conversion is applied, but conversion operators can be created [9].

The reason object modification through inheritance cannot be automatic in Fortran 90, compared to C++, is that objects are not created from modules; they are created from derived types that do not define an IS-A relationship through use-association. In Fortran 90, objects have exactly one type and are not modified by use-association of modules. The base module functions can be used in the derived

module functions, as we have seen, but for a derived module object to have the components of a base module object the base module derived type must be included explicitly in the derived module type of interest. An advantage of this scheme is that a greater mixture of Fortran 90 objects can be created from base modules that contain related derived types.

Inheritance by sub-typing is more restrictive than inheritance by composition. These forms of inheritance are not less effective in Fortran 90 than in C++, but their usage differs. These differences affect abstraction modeling, as will be seen in this chapter and formally examined in Chapter 6. Finally, C++ allows access restrictions to be imposed during inheritance. We already know that Fortran 90 provides access control for modules and derived types, which can be imposed during inheritance as well.

Selective Inheritance One useful feature of use-association is that parts of modules can be used selectively by the `use only` statement. This allows derived modules, or any program unit that uses a module, to select the routines of interest from the base module. This has the benefit that base modules can be very general, but that generality need not be inherited into derived modules. For example, a base module can be created which defines operations involving the charge density and electric fields. However, there may be new additional routines that are specific to a particular field. The general base module routines can be selectively inherited into specific derived modules using the use only statement. Then, the additional operations on the specific fields can be included in the derived module, as seen in figure 4.6.

Now, if new features are required (such as the addition of magnetic effects), operations involving all these fields can be added to the base module, without affecting the definition of the derived modules. Program units which only require usage of specific fields derived from the base module remain unaffected by changes to the base module. In languages where every component of the base class is inherited by the derived

Sketch of Fortran 90 Selective Inheritance

```
MODULE spatialfields_module
IMPLICIT NONE
! interface and derived type declarations...
CONTAINS
   SUBROUTINE solve_fields(cdensity, efield)
   ! field solver routines
   END SUBROUTINE solve_fields
   SUBROUTINE charge_deposit(electrons, cdensity, q)
   ! write to charge density field
   END SUBROUTINE charge_deposit
   SUBROUTINE particle_push(electrons, efield, dt)
   ! read from electric force field
   END SUBROUTINE particle_push
END MODULE spatialfields_module

MODULE chargefield_module
USE spatialfields_module, ONLY: charge_deposit
IMPLICIT NONE
! interface and derived type declarations...
CONTAINS
   ! new specific routines on charge field...
END MODULE chargefield_module
```

FIGURE 4.6: Usage of Fortran 90 `use only` statement for inheriting specific aspects of the base class.

classes the object model would have to be revised. The ability to inherit selectively opens new abstraction modeling capabilities beneficial in scientific programming. Additionally, it promotes safety and extends the lifetime of abstraction hierarchies since features can be added without inadvertently affecting existing components.

Commentary on Inheritance Issues To what extent can Fortran 90 support inheritance? The answer is somewhat controversial, depending on comparison to other languages. For example, a very beneficial feature of C++ is that derived class objects generally can be used as arguments to functions where base class objects are expected because of type conversions. Derived objects in Fortran 90 can access base

module routines, but the typing restrictions of the language do not allow derived module objects to be used where base module objects are expected unless this is constructed explicitly. There is no type promotion as in C++; Fortran 90 objects have exactly one type which must be an exact match for all routine arguments in which they are used.

Inheritance does not exist as an explicit language feature in Fortran 90, but the statements of the language allow for construction of a useful definition appropriate to this language. Using such definitions to copy inheritance in C++ may lead to an awkward program. However, an advantage of use-association of modules with their derived types for inheritance is that a wide variety of Fortran 90 types can participate in the definition of new modules through inheritance. This flexibility allows for beneficial abstract modeling difficult to realize in languages that inherit all features of base classes into derived classes.

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

4.2.4 Generic Programming and Polymorphism

In the Fortran 90 literature, generic programming refers to calling a routine where the action performed depends on the type of the argument. This, perhaps, is not the typical usage for this term. An example of a generic routine in Fortran 90 is the `ABS(x)` intrinsic [11]. The absolute value of `x` is returned by this single call, whether or not it is real or integer valued.

Since the type of every variable must be known at compile-time in Fortran 90, generic routines resemble overloaded functions in languages like C++. Since C++ provides virtual functions—a form of dynamic binding of pointer and reference types

to routines—the terminology "generic" implies that the operation performed varies based on the current address of a pointer. This is possible because base class pointers can also refer to derived class objects in C++. In other words, virtual functions define operations in an inheritance hierarchy based on type. Since the pointer can refer to objects of different types at execution-time, this allows a single pointer to call generic operations on objects which are derived from a common inheritance hierarchy. This form of run-time polymorphism implies that C++ objects belonging to different classes can share identical operations, yet the action of the operation may vary based on the object type.

Fortran 90 supports pointers, but they can only refer to objects of exactly one type. Additionally, pointers are restricted to point to objects that have the `target` attribute; a Fortran 90 pointer may only point to specific type-matching targets. This makes pointers very efficient in Fortran 90, but this restriction means that run-time polymorphic generic operations cannot be modeled in a fashion similar to virtual functions in the C++ language.

Nevertheless, using Fortran 90 *interface blocks*, it is possible to create generic routines which are bound at compile-time. Since such interfaces are automatically provided for routines in modules, when they are part of a generic definition a *module procedure* statement must be specified. For instance, we may want to create scalar and vector fields for charge density and electric forces respectively. Creating these fields in a generic fashion requires the declaration of an interface block in the module calling unit, as seen in figure 4.7. The `fields_create` routine can be called on scalar, vector, and complex field objects in the main program where the proper creation routine will be called since the object is an argument and overloading can resolve the type:

```
TYPE (sfields2d) :: cdensity
TYPE (vfields2d) :: efield
CALL fields_create(cdensity,nx,ny,nxv)
CALL fields_create(efield,nx,ny,nxv)
```

Sketch of Fortran 90 Generic Procedures

```
MODULE fields_module
USE sfields_module
USE cfields_module
USE vfields_module
PRIVATE :: sfields_create, cfields_create, vfields_create
INTERFACE fields_create
  MODULE PROCEDURE sfields_create
  MODULE PROCEDURE cfields_create
  MODULE PROCEDURE vfields_create
END INTERFACE
! rest of module definition...
END MODULE fields_module
```

FIGURE 4.7: Module procedures can be created in Fortran 90 that allow various fields to be initialized by a single function, where the proper initialization routine is called based on the object type.

Since the specific routines, such as `sfields_create` are private, they can only be called using the generic `fields_create` routine. Usage of these internal names is restricted to the fields module.

Polymorphic operations can be implemented in Fortran 90, although the complexity of the construction increases based on their usage. As mentioned, Fortran 90 requires exact matches for routine types, and this applies to pointer types as well. Nevertheless, a dynamic dispatching approach can be applied to support polymorphic object operations similar to virtual function invocations in C++, provided that every object instance is instantiated into a new derived type. This construction models the essence of polymorphic features, although it is not possible in Fortran 90 to construct a single routine that accepts different object types, as in C++. Similar constructions for polymorphism in statically-typed languages are not uncommon; they have been applied, independently, to Ada [56]. Information on dynamic dispatching for polymorphism is available through examples on the Internet [44].

4.2.5 The Object-Oriented Programming Model

Many languages that are object-oriented, or which support object-oriented programming, support some way of mapping the pure object-oriented methodology into a form suitable for practical programming purposes. In Fortran 90, the creation and usage of objects differs from other languages, requiring further consideration.

Fortran 90 objects are created from intrinsic or derived types that are made available by modules. Object declarations that are distinct, yet related, can be grouped into a module where operations involving them are unified through a single interface. The C++ object model of defining operations on a single object can be emulated in Fortran 90 if we restrict a single derived type to each module. However, the ability to include as many derived types as necessary in module definitions opens interesting modeling options. We will see the benefits of the Fortran 90 object model when creating the ternary relation between particles and fields.

The language features of Fortran 90 that can be used to support object-oriented programming have been described. The new statements are also useful in more traditional programming paradigms, as is suggested in most Fortran 90 texts. We believe, however, that the abstraction modeling concepts introduced will encourage a paradigm shift useful in scientific application development.

4.3 Plasma PIC Application Programming

Now that the fundamental features of object-oriented programming in Fortran 90 have been presented, we can consider their usage in plasma particle-in-cell programming. Since we have already discussed the process of designing object models, as well as the features of this application, we will immediately focus our attention strictly on the programming issues.

The Fortran 90 programs were developed in a collaborative manner. Some of the most recent scalar programs were extended by Viktor Decyk based on conversations

and the orignal C++ codes. While many of the Fortran 90 program models are similar to the C++ versions, and vice-versa, the reasons for these differences are based on language features that affect abstraction modeling. This important issue will be addressed in Chapter 6.

4.3.1 Fortran 90 Mirror of C++ Model

Returning to the original one-dimensional C++ scalar program of figure 3.3 on page 36, our interest was to mirror every feature of this program in Fortran 90. By establishing if the new constructs could model an existing C++ program, we could examine Fortran 90's abstraction modeling capabilities and performance.

Modules in Fortran 90 were created to represent the classes of the C++ program. Objects were created from the derived types within the modules by use-association, where the modeling hierarchy matches that of figure 3.3. Since constructors are provided in C++ we emulated them in Fortran 90 by including `module_create` routines, which were called to initialize objects. The array syntax operations in Fortran 90 were very useful; they replaced a number of iterative loops in field operations used in the C++ version.

The Fortran 90 main program is shown in appendix A.11 on page 203. This should be compared to the C++ program in appendix A.6 on page 195. Although some of the naming conventions are different, they are essentially identical. It is also interesting to compare the Fortran 90 main program to the one-dimensional Fortran 77 initialization and loop sections of appendix A.1 and appendix A.2 on page 190 and page 191 respectively. Many of the inner details of the Fortran 90 program are similar to the original Fortran 77 version, but the interface to these details is very different.

4.3.2 Program Organization Based on Fortran 90

The Fortran 90 scalar one-dimensional mirror program was a copy of the C++ version. Fortran 90 features were not used based on their capabilities, but rather based on

the features of C++. This section presents a new program organization based on the features of Fortran 90, developed in collaboration with Viktor Decyk. The new model was more physics-oriented than the original one-dimensional C++ version, given that we could represent these abstractions more directly with modules and derived types. (The more advanced C++ programs already presented were developed concurrently with the Fortran 90 programs, so components of the following designs should seem familiar.)

The organizational changes were motivated by the features of modules, derived types and use-association. Many distinct, but related, items can be included in a single module using derived types. The objects created by use-association represent specific parts of the module based on the derived types contained within. This provides a very "open" modeling strategy.

One-Dimensional Sequential Program The modified one-dimensional sequential program contains modules for the distribution of particles, energy diagnostics, particle species, fields, collective plasma, and related operations much like similar models already presented. The interesting part of the Fortran 90 version is the organization of these components. The `fields_module` contains a derived type for dynamically creating scalar fields with field operations. The fields created include the one-dimensional charge density and electric field, with operations to solve field equations in complex Fourier space. Since all of these fields come from the same module, however, their components are immediately accessible so the numerical routines can operate on them directly. This allows the module to contain two very different yet related constructs, charge density and electric fields, where all operations involving the fields, individually or collectively, are unified in a single place and accessible by use-association.

In Fortran 90, a `species_module` may be defined which includes the definition of a one-dimensional particle with other aspects of the species, as we have seen in

figure 4.2 on page 71. The previous codes separated the definition of a particle from features that describe the collection of common particle properties. The derived type, combined with the structure of modules, supports this view naturally.

Although building an equivalent species class in C++ is possible, the benefit of the module organization can be seen in extensions to mixed-dimensional codes. Higher-dimensional particle types can be added directly to the module, so a single code can be written to support higher-dimensional simulations by use-association of the original module. This could encourage the development of module libraries that can be shared among related codes.

The `plasma_module` can easily represent the unification of all components that make up the simulation (energy diagnostics, particle species and fields), by simple use-association of these components:

```
MODULE plasma_module
USE energy_module, species_module, fields_module
IMPLICIT NONE
CONTAINS
   ! plasma simulation routines...
END MODULE plasma_module
```

In this manner, a simple form of multiple inheritance is applied. All routines that involve these components can be unified into the plasma module, so usage of the module in the main program makes every routine and type immediately available. This is seen in appendix A.12 on page 205, where the class hierarchy is shown in figure 4.8. Many of the modules contain derived types as part of their definition, such as the definition of a `particle` in the species module, as previously seen.

Higher-Dimensional Sequential Program Organization The latest C++ codes used templates to combine related field components into a single structure. When considering the design of the next generation of Fortran 90 codes, even though Fortran 90 currently does not support parameterized types, this grouping of field elements was used since it also made sense physically. Now, the charge density field would be

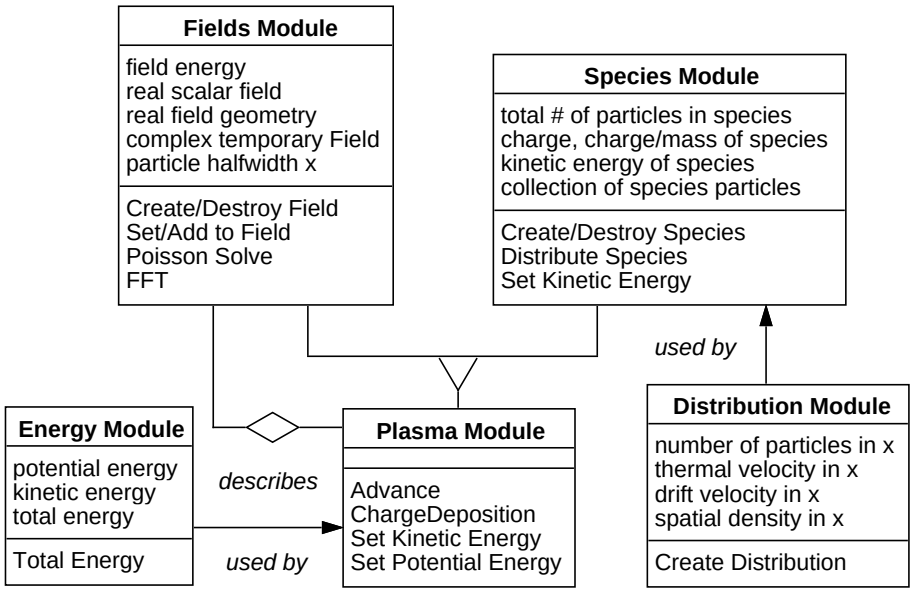


FIGURE 4.8: Fortran 90 one-dimensional scalar class hierarchy (OMT notation). This design, derived from an early C++ version based on Fortran 90 constructs, unifies particle species and field in the definition of plasma simulation operations.

a scalar field while the electric field would be a vector field. This nomenclature was introduced somewhat independently into the Fortran 90 codes, later finding its way into the latest C++ versions. Also, since the field solvers operate in complex space, a new complex fields module was introduced to represent the fields in this space since their geometry and purpose is quite different from scalar and vector fields. To support the conversion between the complex and scalar/vector fields, a fields module was also added. This module is responsible for solving the field equations using objects of the three field types: scalar, vector and complex.

Part of the vector field (vfields_module) is shown in figure 4.9. The module structure allows the components that comprise the definition of the vector field to be included in the module. Creating a pointer to a dynamic two-dimensional array which represents the vector field is simple, shown in the modification of the `dimension` attribute. Fortran 90 can support dynamic allocation of arrays in many dimensions. All array intrinsic operations are automatically supported.

Sketch of Fortran 90 Vector Fields Module

```
MODULE vfields_module
IMPLICIT NONE
  TYPE point2d
    real :: x, y
  END TYPE point2d
  TYPE vfields2d
    real :: wp             ! field energy
    integer :: nx, ny      ! field geometry
    type (point2d), dimension(:,:), pointer :: p
  END TYPE vfields2d
CONTAINS
  SUBROUTINE vfields_create(this, nx, ny)
  TYPE (vfields2d), INTENT(inout) :: this
  INTEGER, INTENT(in) :: nx, ny
    allocate (this%p(nx,ny), stat=ierr)
    ! rest of subroutine body...
  END SUBROUTINE vfields_create
  ! additional routines...
END MODULE vfields_module
```

FIGURE 4.9: Structure of a Fortran 90 two-dimensional dynamic vector field module.

Modifications to the field definition represent the major difference from the one-dimensional code. A sketch of the main program is shown in appendix A.13 on page 207. The new model, extended from the one-dimensional model of figure 4.8 can be seen in figure 4.10.

4.4 Parallel PIC Programming in Fortran 90

To learn more about how object-oriented techniques in Fortran 90 could be applied to parallel programs, we developed a two-dimensional model as an extension from the scalar program. Although High Performance Fortran (HPF) compilers are becoming more available, currently most of them only support the subset-HPF standard and many do not support important Fortran 90 features, such as modules. The MPI mes-

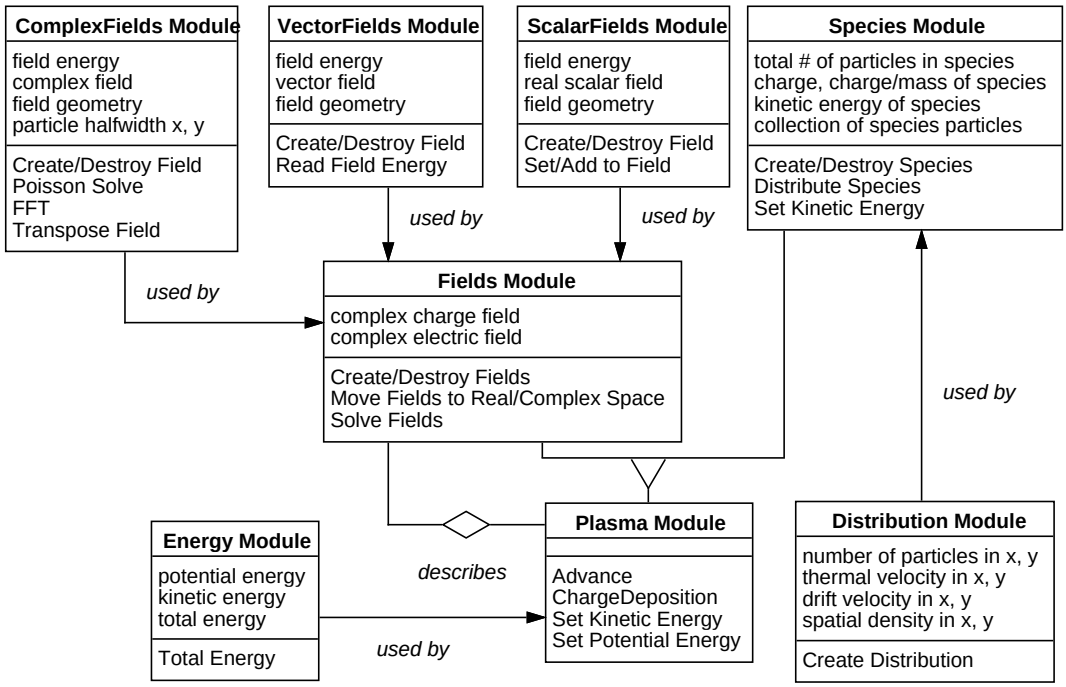


FIGURE 4.10: Fortran 90 two-dimensional scalar class hierarchy (OMT notation). Individual field types form components of the complete field. The plasma module routines operate on the unification of the particle species and various field components.

sage passing interface standard was used for parallelism. This required modifications to the MPI header files, mainly adding interface declarations for some routines, since we used the Fortran 77 compiled library calls in Fortran 90.

Extending the Fortran 90 two-dimensional hierarchy of figure 4.10 on page 92 was unusually straightforward and direct. Fortran 90 array subsection operations were beneficial in guard region operations. Use-association also simplified introducing MPI routines into the program. An `mpi_module` included the library routines, local data, and application specific functions useful in message passing. The components of the module were accessible to any program unit involved in message passing by use-association:

```
MODULE mpi_module
  INCLUDE 'mpif90.h'
  IMPLICIT NONE
```

Sketch of Partition Module with MPI Usage

```
MODULE partition_module
  USE mpi_module
  IMPLICIT NONE
  TYPE slab
    ! definition of slab partition...
  END TYPE slab
CONTAINS
  SUBROUTINE dcomp2(edges,ny)      ! perform partitioning
    TYPE (slab), INTENT (out) :: edges
    INTEGER, INTENT (in) :: ny
    ! remainder of routine...
  END SUBROUTINE dcomp2
END MODULE partition_module
```

FIGURE 4.11: Sketch of a partition module used in the Fortran 90 parallel program. MPI routines are encapsulated in a module made available for partitioning operations by use-association.

```
    INTEGER :: idproc, kstrt, nproc, nprec
  CONTAINS
    SUBROUTINE ppinit(nvp)       ! initialize for parallelism
    INTEGER, INTENT(in) :: nvp
      INTEGER :: ierror
      call MPI_COMM_RANK( MPI_COMM_WORLD, idproc, ierror )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nproc, ierror )
    END SUBROUTINE ppinit
    SUBROUTINE timera(icntrl,chr,time)
      ! details of timer using MPI routines...
    END SUBROUTINE timera
  END MODULE mpi_module
```

The `include` statement makes standard MPI calls available where the local variables are also accessible when the module is used, such as in the `partition_module`, seen in figure 4.11, or any other module that requires message passing.

The C++ programs included partition objects in the definition of classes that operated on partitioned data, such as the particle species and fields. In Fortran 90, we

can simply make partition information "available" by use-association where needed, such as in the `species_module`. In this instance, we are simply providing partitioning information as static member data with the `use` statement, there is no inheritance relation.

```
MODULE species_module
  USE distribution_module
  USE partition_module
  ! definition of derived types
CONTAINS
  SUBROUTINE species_distribute(this,edges,distf,noff)
  TYPE (species2d), INTENT (out) :: this
  TYPE (slab), INTENT (in) :: edges
  ! remainder of routine...
END MODULE species_module
```

In the creation of scalar and vector partitioned fields, however, the usage of the partitioning module does represent inheritance since it is used in partitioning the field, not simply to find border information.

The module hierarchy is illustrated in figure 4.12 with the main program in appendix A.14 on page 209. The derived types, from which objects are created, are contained within the modules. The differences from the hierarchy of figure 4.10 on page 92 are minor. Many classes developed for the C++ version to support multi-dimensional dynamic arrays and array operations are not required since Fortran 90 supports this in the language itself.

4.5 Commentary

It is important not to confuse object-oriented methodology with its realization by a particular language, such as C++ or Fortran 90. The specification of the paradigm is language independent. Some languages, like C++, provide support for object-oriented programming that extend beyond the formal description of the paradigm,

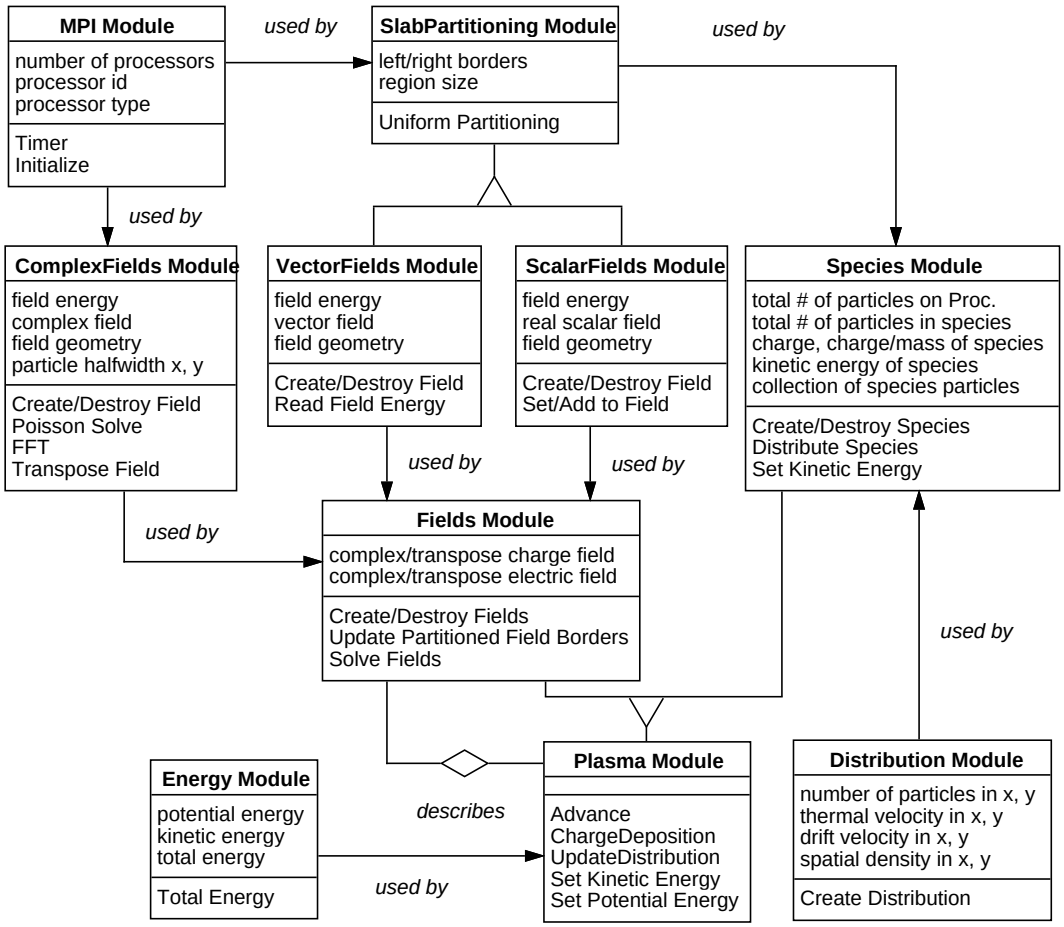


FIGURE 4.12: Fortran 90 two-dimensional parallel class hierarchy (OMT notation). The sequential version was extended by adding a module to make MPI communication routines visible as well as the inheritance of partitioning information into distributed fields.

such as templates. Some of these features go beyond basic language statements such as automatic type conversion and the ability to choose whether a base class or derived class member function should be called in various instances.

There are many popular aspects of object-oriented programming that have not been discussed in the context of Fortran 90, such as templates, run time type identification (RTTI), design patterns, and so on. Some features are currently in discussion within the Fortran 90 standards committee, such as parameterized derived types which are similar to templates. At this point in time, many aspects of object method-

ology are still in the research stage; it may be premature to formally introduce them into a language so dominant in scientific programming as Fortran. We do not wish to imply that compiler designers should rush to add features to convert Fortran 90 into a language which is more object-oriented, possibly to increase its popularity or to formally proclaim Fortran 90 as an object-oriented language. Our view is that Fortran should continue to evolve, as it has in the past, based on the experience with other languages.

There are benefits in applying the object-oriented methodology to languages that, strictly speaking, lack some features to exploit this methodology directly. One of the benefits of applying this technique to Fortran 90 is that entities that conceptually belong together, but may not share a hierarchical relationship, can be modeled in this language easily. Such capabilities have also been useful in the features of C++ that are not "purely" object-oriented [61]. While some features, such as the IS-A relationship seem to be a hallmark of general-purpose object-oriented modeling of "real world" concepts, we have found the HAS-A form of composition inheritance more useful in modeling physics concepts than the sub-typing inheritance model.

Object-oriented techniques have been applied to languages not known to support this methodology before, somewhat notably with the Ada language [3, 56] as well as other languages [55]. Indeed, we recently found the Seidewitz paper and were surprised at the similarities in approach, compared to our methods, given the differences in Ada and Fortran 90. One of the exciting aspects of the new Fortran 90 constructs and their usage is that Fortran is very popular and the new techniques can be applied incrementally to existing software. The object-oriented techniques need not be used *duplicate* techniques in other languages, such as C++, rather the modeling methods can stand on their own. In practice the emulation of C++ statements in Fortran 90, such as sub-typing inheritance, may require an effort much larger than most designers may tolerate.

As suggested, many of the features of Fortran 90 are beneficial even if the object-

oriented approach is not fully adopted. Nevertheless, the methodology can establish a context through which these new features may be applied. Finally, while it is possible to use Fortran 90 constructs to model aspects of C++, this can be tedious at times. In particular, it is important that inheritance is not automatically part of the Fortran 90 language standard; it must be constructed. However, the flexibility of the language allows for important new modeling concepts to be explored. In [60], it is suggested that languages *support* a style of programming based on the ease, convenience and safety by which the features of that style are supported. Excessive effort or ability required in modeling this style are good indications that the language only *enables* various techniques to be applied. Our recommendation is to use modules and derived types in abstraction modeling, with judicious use of use associations and other advanced language features, to develop a programming style appropriate to the semantics of Fortran 90, not necessarily other languages. This has been our approach in application development and it has been extremely successful.

Chapter 5

Irregular Computation in Plasma Modeling

This chapter describes irregularity in plasma computation and discusses how object-oriented techniques can be introduced to address this problem. The forms of irregularity in plasma simulation are unusual and quite different from other applications given the structure of the PIC algorithm. Since modification of simulation parameters can cause experiments to require load balancing, handling irregularity is necessary for extension to new problems. An implicit monitoring technique, which is an object-oriented approach used in instrumenting programs to address load imbalance, is described in the context of the collisionless free-expansion and gravitational experiments.

5.1 An Object-Oriented Approach for Irregular Computation

Modern simulation of various phenomena in the physical sciences and engineering has become increasingly dependent upon distributed memory parallel computation. Many applications are *regular*; the computation workload distribution remains static (balanced) across the processors during execution. More sophisticated problems require

a dynamic redistribution of processor workload to offset performance degradation. These *irregular* computations are difficult to implement since dynamic load balancing is required. The form of irregularity is usually problem dependent. Furthermore, current programming techniques are not designed for problems of this kind, which are among the most challenging to model on distributed-memory parallel machines. This implies an interdependence between the programming methodology and the techniques used to maintain efficiency for irregular scientific computations.

Our approach in addressing this issue involves a programming paradigm and mechanism based on *continuous implicit monitoring* of program characteristics via class abstraction hierarchies. Such an approach can allow for dynamic load balancing through a monitor class interface whereby program development can be separated from load balancing implementation issues. This paradigm extends object-oriented methods beyond program abstraction, into a methodology useful in addressing irregular scientific programming problems.

In the approach, data collected for run-time analysis is implicitly acquired through a special monitoring class inherited into an application class hierarchy. Operations performed on user-defined derived classes trigger implicit (hidden) operations which collect information used by the run-time load balancing routines. Additionally, the monitor itself makes new capabilities explicitly available to derived classes for run-time analysis of performance characteristics used for load balancing computational objects. Since user defined objects can also be viewed as instances of the monitoring base class, dynamic load balancing can be performed on these objects through monitor class operations. The monitoring methods are continuous, allowing for use of these techniques in applications that require frequent load rebalancing, such as in certain simulation problems. This encapsulates the load balancing mechanisms while separating them from other application implementation issues. An important concern involves selection or design of methods appropriate for continuous monitoring. The advantage of such an approach is that various load balancing algorithms can be

introduced on top of the mechanism that provides information about the run-time characteristics of the simulation. As properties of the simulation change in an irregular fashion, different load balancing methods may potentially be applied within the same object-oriented context to improve efficiency based on trend information discovered within the application.

As mentioned, programs are instrumented for monitoring implicitly since measurements are associated with object usage. Because of the inheritance based usage of monitors, virtual functions in C++ allow generic monitor operations to behave differently based on the derived object type. In this way, load balancing methods that operate on monitors can manipulate program objects differently based on their type. Before we introduce this method, which is most clearly described by example, we must understand the context in which it will be applied.

5.2 Computation and Communication Irregularity in PIC Simulation

When dynamic load balancing is applied in the General Concurrent Particle-in-Cell (GCPIC) method, particles remain relatively well balanced across the processors as they move among partitioned regions. Since all particles must deposit charge through interpolation to the charge density field, as well as acquire forces through interpolation from the electric force field, both of these fields must vary in size with the density distribution of particles in the spatial region. The field equations are solved in Fourier space, which requires a *uniformly partitioned* field across the processors. This complicates the simulation since load balancing not only includes balancing the particles by moving the charge density and electric force field boundaries, but also includes mapping these non-uniform fields back and forth between the uniform Fourier fields. The nature of this coupled mapping introduces irregular data and communication which for highly dynamic problems may occur every time step.

As suggested, there are a number of fields that affect the motion of the particles. These include the scalar charge density field, vector electric field, and the complex Fourier field. When these fields are uniformly partitioned and static, mapping from the charge density to Fourier to electric field is straightforward since load balancing is not required. Consequently, only the guard regions of the scalar and vector fields must be exchanged with their neighbor processors; the major portion of the fields can be mapped locally. In fact, this uniformity allows each processor to easily map between field regions in stages listed in figure 5.1. The regularity in communication

Uniform-Static Partitioning Mapping Algorithm

1. Write charge density to local Fourier field
2. Send left guard region to left neighbor
3. Send right guard region to right neighbor
4. Solve field
5. Write Fourier field to local electric force field
6. Send left Fourier region to left neighbor
7. Send right Fourier region to right neighbor

FIGURE 5.1: Partition mapping when fields are static and uniform.

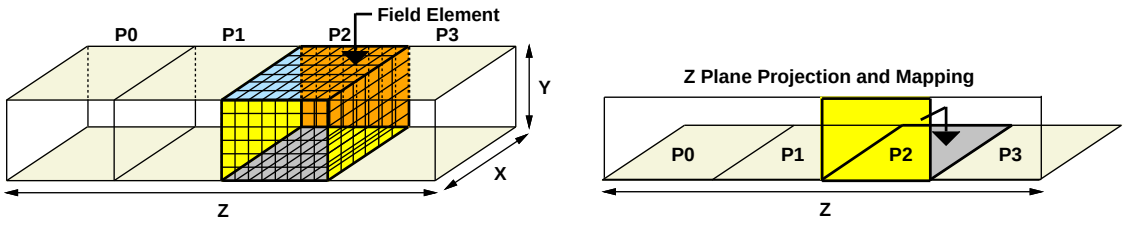
and computation using this approach is apparent. The technique is also very efficient, since the number of guard regions transported to neighbors is small compared to the total field size.

This section will study the details of how this method must be modified when moving boundaries are involved. We will see that an algorithm for this problem involves a number of issues that are not immediately apparent. Our goal was to design a time-space efficient algorithm which minimizes the communication overhead. After the technique used in managing particle and field partitioning has been presented, we examine the modifications to the plasma PIC algorithm for load balancing. Subse-

quently, the monitoring method is introduced for load balancing the free expansion of plasma in a vacuum and for a gravitational problem.

5.2.1 Managing Particle-Field Partitioning

Since the field is partitioned into subfields consisting of uniformly distributed grid elements of finite size, the partition borders will be resolved to the integral address of a grid point normal to the partitioning direction. In this problem, it turns out that a slab-partitioning in three-dimensions, rather than rod or cube partitions, is sufficient since maintaining more refined partitioning would overwhelm the time spent in load balancing [32]. In figure 5.2, we show the field on processor P_2 in three-dimensions (c.f. 5.2(a)). Since slab-partitioning in the Z-coordinate will be used, we can simplify the mapping illustrations by drawing the two-dimensional projection normal to the Z-plane, hiding the XY-Plane. Additionally, this projection will be re-mapped into the page so that mappings between different fields can be drawn in a single illustration (c.f. 5.2(b)).



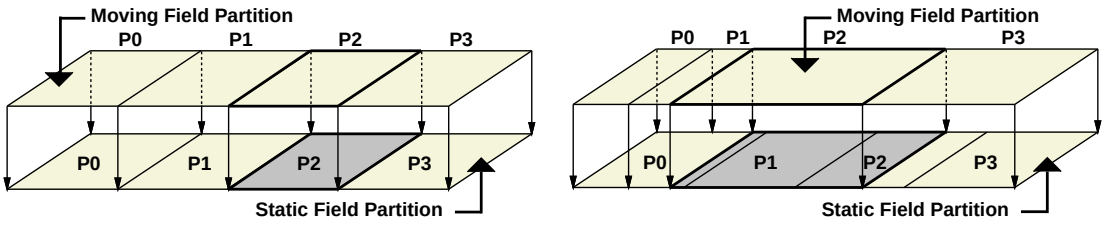
(a) Three-Dimensional Field

(b) Field Projection Re-mapping

FIGURE 5.2: Illustration of slab partition mappings used to represent field partitioning. Projections of three-dimensional fields into the plane are shown.

Partitioning examples are shown in figure 5.3. A uniform partitioning (c.f. 5.3(a)), where the Fourier and particle fields are identical, and a non-uniform partitioning (c.f. 5.3(b)), where the particle fields are variant, are illustrated. The shaded region indicates how partitions overlap when processors map field regions into the complex

uniform Fourier fields.



(a) Uniform Partitioning (b) Non-Uniform Partitioning

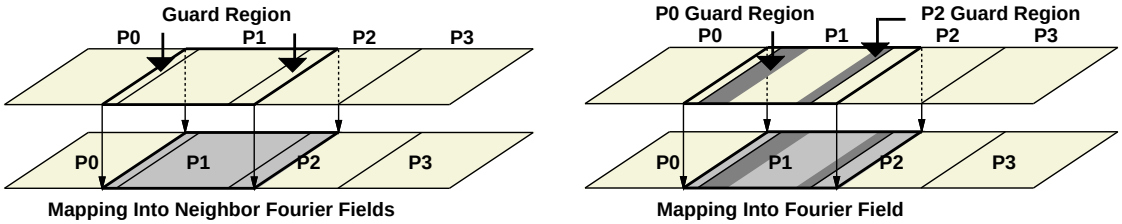
FIGURE 5.3: The uniform partitioning only requires transmission of the guard regions (not shown), while the non-uniform partitioning may require more extensive interprocessor communication beyond the guard region transfers. For instance, P_2 communicates with P_0 and P_1 in addition to itself.

As indicated, the moving partition and static partition regions belong to distinct processors. They can overlap in arbitrary ways, with some restrictions. We have already mentioned that partition borders will only be resolved to a grid point. Additional assumptions on the position of partitions include the following:

- Borders shared by neighbors have the same global address.
- Borders (in general) cannot overlap. The guard cells in moving partitioned regions are an exception, but they only overlap in a *logical* way, not in terms of their addressing.
- The external boundary borders on processors P_0 and $P_{(n-1)}$ are fixed, where n is the total number of processors.

We do not claim that the number of moving partitions segments must equal the number of static partition segments, nor that they must equal the number of processors. In most cases, however, these circumstances will hold.

Although the images shown in figure 5.3 are two-dimensional, the fields partitioned are three-dimensional. In figure 5.4 (c.f. 5.4(a)), we illustrate how the guard region



(a) Local Guard Cell Mapping

(b) Neighbor Guard Cell Mapping

FIGURE 5.4: Examples of field and guard cell mappings among processors into the Fourier partitions. The Fourier partition is in complex space and does not maintain guard planes.

mapping on processor P_1 extends into static partitions on neighboring processors (P_0, P_2). In the accompanying figure (c.f. 5.4(b)) we see how the guard regions in neighboring processors (P_0, P_2) map into the local field region on P_1. There are two guard regions on the right and one on the left, but this can vary based on the interpolation scheme used. Managing the interaction among these regions will be an important part of the partitioning scheme.

Rotating Field Partitioning Approach On a message passing machine inter-processor communication is required to send and receive field components among processors that need them. Therefore, processors must know which portions of the global field they own in addition to those regions required by other processors. To guarantee that each processor will receive the field sections required to perform Fourier transforms one simple approach is to rotate the field components among all the processors. As fields arrive processors select the portions required, sending the incoming field to its neighbor. While straightforward to implement, the communication cost of transmitting the entire field among processors is generally unacceptable.

Extended Guard Cell Partitioning Method Another partitioning approach is to extend the static field transfer approach for moving fields. Since the static

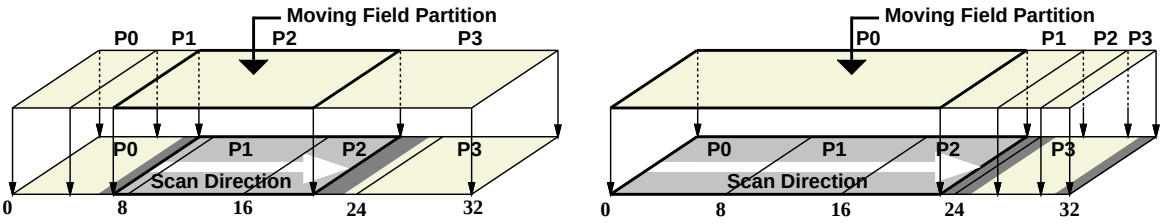
method sends guard data to the left, writes local data, and sends guard data to the right, moving partitions that extend beyond static partition borders can be viewed as "extended" guard cells. The approach would require processors to determine if they have extended guard cells, as well as if local writes must be performed. Then, the send-left, local-write, send-right paradigm could be preserved. Tests on the partition borders that would be performed include:

```
if (MovingPartitionLeft ≤ StaticPartitionRight &&
    MovingPartitionRight ≥ StaticPartitionLeft)
    // Perform Local Write...
if (MovingPartitionLeft ≤ StaticPartitionLeft &&
    MovingPartitionRight ≥ StaticPartitionRight)
    // Extended Guard Cells Method...
```

Actually the tests performed are much more sophisticated than this, as is the action taken when these tests pass or fail. Although this approach can work a more consistent method is desirable. An implementation based on the aforementioned approach might wait unnecessarily if there are no guard regions to send. Additionally, special cases must be evaluated. What happens, for instance, when the range of the moving partition borders of processor P_k does not include P_k as a subset? We do not perform a local write, but it may not be correct to think of the transfer as an extended guard cell operation.

The Scanning Partition Algorithm The method that we have introduced for mapping from the moving to the static partitions is based on scanning. A moving partition is scanned from left to right in the dimension that is to be partitioned. During this scan, field data is collected and packed into a buffer. The amount and destination of this data depends on the processor identification number of the partitions scanned, their dimensions, the location of the partition borders, and the number of guard planes.

Figure 5.5 shows typical scan examples for a selected moving partition. Notice how the moving partition must map its field region across processors, as well as



(a) Processor P_2's field region extends over P_k's ($k = 0, 1, 2$) Fourier region.

(b) Processor P_0's guard region extends beyond defined border on P_2, causing interaction with P_3.

FIGURE 5.5: Various scenarios illustrating issues that the scanning algorithm must address. The guard cell regions in the moving partitions complicate the scanning technique.

its additional exterior guard regions (c.f. 5.5(a)). (These exterior guard regions are only illustrated in the static partition mapping, not in the moving field partition mapping.) Also, recognize that the left guard region on processor P_0 (c.f. 5.5(b)) also causes interaction with processor P_3 since a periodic field mapping is used. The scan address is based on the global location of the partition borders, where sample indices are shown in figure 5.5. The global address of the guard regions will not be scanned, but they are accountable during the scanning process. In the rest of this section, we discuss the algorithm and the implementation details.

5.2.2 The Scanning Partition Mapping Method

In the mapping technique there will be an exchange of information between the moving and static partition regions across processors. For instance, before a scan can be performed, since the data is distributed, the static regions must know how many receives to post for the incoming field segments. Only the moving regions can compute how many sends to perform to the appropriate processors. Therefore, these moving regions must determine the location of their borders with respect to the static partitioning, the number of sends required (including guard cell operations), and the

number of receives to post on behalf of the static regions. A global view of the scanning partition mapping method can be seen in figure 5.6.

Scanning Partition Algorithm[a]

1. Each processor inspects its moving partition boundaries and determines the processor(s) whose static field coordinates contain these boundaries.
2. Each processor performs a scan from "left to right" across its moving partition. Iteratively pack field data and information required for the inverse mapping—from the static to moving partitions—and transport as a contiguous message to the static partitions that require this information.
3. Each processor that owns a static partition, using an inverse mapping table dynamically constructed, iteratively maps field data back to the moving partitions.
4. Repeat the process in the next time step.

[a]Details are explained in the subsections.

FIGURE 5.6: Sketch of the Scanning Partition Mapping Method.

These stages, while apparently straightforward to describe, actually have sophisticated components that require complex programming within a distributed memory message passing environment. They shall be examined independently for clarity, but the algorithm relies upon the transfer of information between stages.

Location of Processor Partition Borders (Step 1) Every processor must determine the relationship between its moving field partition borders and the borders belonging to processors in the static partitioning. Without this information, it is impossible to map the moving fields to the static fields since there is no means to determine where field data is sent, and received. Fortunately, the Fourier field must be uniformly partitioned, so mathematical short-cuts can be introduced in this stage. Unfortunately, the guard cells complicate matters since they can introduce new interactions between processors due to the current partitioning, as we have seen in figure 5.5. Additionally, information required by future stages for interprocessor

communication must be collected as these border locations are determined.

The uniformity of the Fourier field allows the location of the moving borders on any processor to be computed immediately:

$$\begin{aligned} \mathrm{PE}_L &= \lfloor \mathrm{MP}_L/\mathrm{SPSIZE} \rfloor \\ \mathrm{PE}_R &= \begin{cases} \mathrm{PE}_{n-1} & \text{for processor } (n-1) \\ \lfloor \mathrm{MP}_R/\mathrm{SPSIZE} \rfloor & \text{otherwise} \end{cases} \end{aligned} \tag{5.1}$$

where (PE_L, PE_R) are the processors that own the static regions where the moving region borders fall, (MP_L, MP_R) are the global indices of the (left, right) moving partitions, SPSIZE is the fixed size of the partitioned Fourier field region on each processor and n is the total number of processors indexed from 0 to $(n-1)$. Note that the external borders on processors P_0 and P_{n-1} are fixed. The values of PE_L and PE_R are used as bounds when computing the range of static partition regions to which the moving regions are mapped on each processor. This mapping implies interprocessor communication, in particular:

1. Every processor that owns a static region *must know* how many field communication messages to expect from processors that own the moving regions.

2. Similarly, every processor that owns a moving region *must know* how many field communication messages to expect from processors that own the static regions.

These are important points which cannot be overemphasized. To address these points, the interaction of the guard regions with the static partition borders must be considered.

In determining the number of receives posted by the static regions, given the location of the moving region borders (case 1), one soon realizes that *processors which own static regions cannot compute this number.* The processors responsible for the static regions have no information about the location of the moving borders, however,

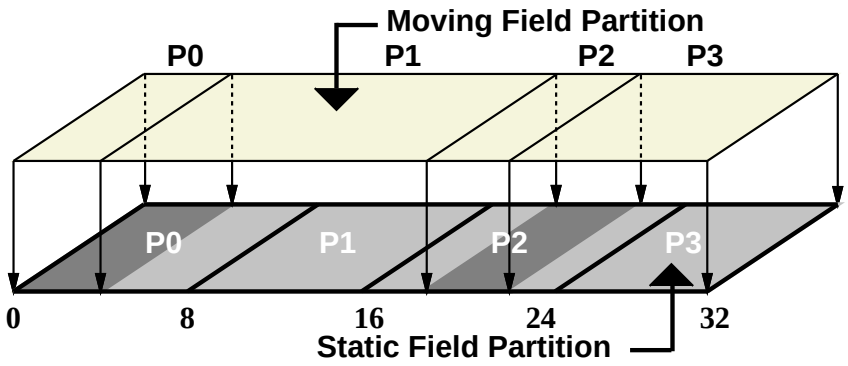


FIGURE 5.7: Sample mapping in static partition receive posting algorithm.

the processors responsible for the moving regions do know that the static partitioning is uniform. Using figure 5.7 as a canonical example, we show the mapping of moving partition regions into the static partition regions. For instance, P_1's moving partition region induces communication with the static partition regions on processors P_0, P_1 and P_2. Including self-communication in these computations is convenient, as will be seen when the message passing protocol is examined.

The number of receives posted by processors that own the static regions can be computed by the parallel algorithm of figure 5.8. Determining when moving boundaries share the same address as static boundaries can be found by extensions to equation 5.1. This involves ensuring that the arithmetic is performed carefully. The exterior boundaries of the simulation space are fixed, so this condition is handled as a special case, as are the effects of the guard regions. The conditional testing is sophisticated, but manageable.

Table 5.1 shows the result of the computation performed by the static partition receive-posting algorithm for figure 5.7. For example, processor P_0 maps into static regions on processor P_0 and P_3 (from the periodicity of the fields and the guard regions at address zero), so this is indicated in the first row of the table. Similarly, processor P_1's moving region maps into the static regions of processors P_0, P_1 and P_2, which are also marked. Each processor simultaneously makes these marks into a shared

Static Partition Receive–Posting Algorithm

1. Mark the processor number for *exterior* and *interior* moving guard region borders if the moving and static partition borders share the *same* address.
2. Mark the processor number which owns each static region that is covered between the moving partition border regions (also implies a static partition receive).
3. Perform parallel summation of marked identifiers to compute the number of receives to post for the static region owned by the current processor.

FIGURE 5.8: Sketch of receive-posting algorithm for static partitioning.

array where each index corresponds to a static partition processor. Then, a parallel summation over the array is performed to determine the number of receives that the processor owning a static region must post. The result is shown in the second row of table 5.1. Although the static region owned by each processor now knows how many

Action	P_0	P_1	P_2	P_3
(moving) Processor(s) mapping into (static) P_k	0, 1, 3	1	1, 2, 3	0, 3
Number of Receives Expected by (static) P_k (σ_k)	3	1	3	2
Number of Receives Expected by (moving) P_k (μ_k)	2	3	1	3

TABLE 5.1: Receive-Posting algorithm computation example. The first row indicates how P_k's moving region maps into the static partition regions of the processors listed. The second shows the number of receives expected by processor P_k which owns the static partition region. The last shows the receives posted by processor P_k which owns the moving partition region.

receives to post, *none of these processors know where the field messages will originate.* Fortunately, the receive order can be arbitrary, so processors sending regions they ow can include their identification number in the message. This information is required for the inverse mapping of the static regions back to the moving partition regions, however, additional data must also be included.

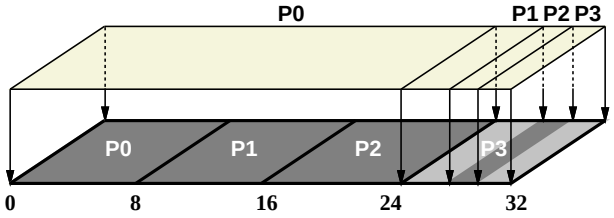
The computation of the number of receives that processors owning static regions

must post can yield additional information required when performing the inverse mapping. The number of sends performed by the moving regions are exactly the number of receives these regions must post in the inverse mapping, illustrated in the third row of table 5.1. For example, since there are two zeros in the first row, indicating that P_0 sends to P_0 and P_3, two sends will be performed. Similarly, processor P_2 only performs one send—it communicates with itself–but this cannot be assumed in the forward mapping. Processor P_1 maps into P_0, P_1 and P_2 while processor P_3 maps into P_0, P_2 and P_3 indicating three sends each.

The sample partitioning of figure 5.7 on page 109 does not induce additional interactions with neighboring static regions. Since the partitioning may be arbitrary, as seen in figure 5.9, the algorithm must also handle this case. An analysis of the receive posting algorithm has been performed [43]. The algorithm is linear in the number of processors due to the number of messages that may be transmitted.

The Scanning Stage: Mapping Moving Fields to Static Fields (Step 2) The general operations performed in this stage are sketched in the scan-pack-move algorithm in figure 5.10 on page 113. Generally, each processor owning a moving partition begins scanning and packing its field from its leftmost border until either a static partition address, or the rightmost moving border, is encountered. Guard regions are included as part of the contiguous message based on testing a variety of conditions involving the moving and static border locations and the current location of the scan index. When the proper conditions are satisfied, the region scanned is transported to the appropriate destination processor—which expects a message due to the receive-posting algorithm. This technique divides the moving field regions into blocks based on the uniform static partitioning. The process repeats until all blocks that comprise the moving field region have been transported.

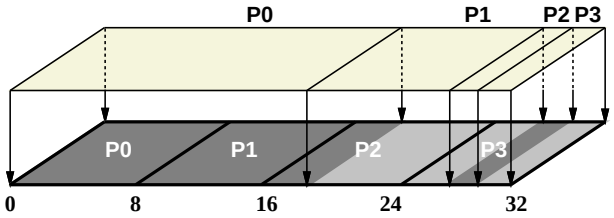
A example of this process on a moving field component is illustrated in figure 5.11. When processor P_2's moving region performs the scan (c.f. 5.11(a)), it must pack its



(a) Multiple interaction of P_0 with P_3.

	P_0	P_1	P_2	P_3
Map	0, 3	0	0, 1	0, 0, 1, 2, 3
σ_k	2	1	2	5
μ_k	5	2	1	2

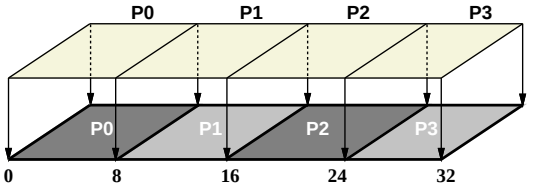
(b) Receive Posting Table



(c) P_1 never communicates with itself.

	P_0	P_1	P_2	P_3
Map	0, 3	0	0, 1	0, 1, 2, 3
σ_k	2	1	2	4
μ_k	4	2	1	2

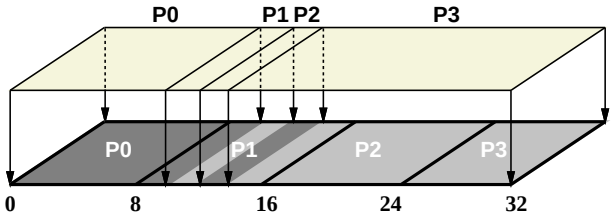
(d) Receive Posting Table



(e) Maximum level of communication.

	P_0	P_1	P_2	P_3
Map	0, 1, 3	0, 1, 2,	1, 2, 3	0, 2, 3
σ_k	3	3	3	3
μ_k	3	3	3	3

(f) Receive Posting Table



(g) Counterexample that $\sigma_k \neq (\mu_k)^R$.

	P_0	P_1	P_2	P_3
Map	0, 3	0, 1, 2, 3	3	0, 3
σ_k	2	4	1	2
μ_k	3	1	1	4

(h) Receive Posting Table

FIGURE 5.9: Sample partition mappings showing various cases that the mapping algorithm must address. The guard regions inducing interprocessor interactions are not shown explicitly.

Scan–Pack–Move Algorithm

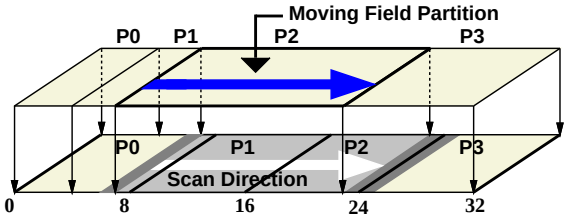
1. Initialize scan index to leftmost border of moving partition region.
2. Given the current scan index, compute next static border index.
3. Determine if field is packed to the next static border index or to the rightmost border of the moving partition region.
4. Pack field data to the correct border and transport to appropriate processor. (Include information required by the inverse mapping.)
5. Continue scan-pack-move operations (from step 2) until the rightmost border of the moving partition is reached and the field is transported.

FIGURE 5.10: Sketch of algorithm used to map moving field partitions to static field partitions across processors.

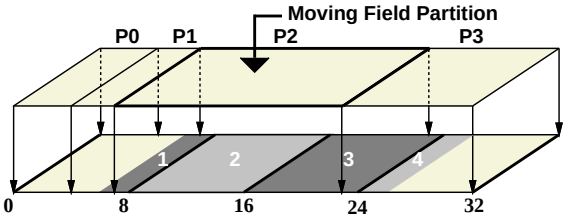
left guard region for processor P_0 and additional information required by the static region on P_0. Then, the region of the moving partition on P_2 that is mapped onto the static region owned by P_1 is packed. Next, the region on P_2 mapped onto itself is packed, including part of the guard region. Finally, the last portion of P_2's guard region is packed and transported to the static region owned by processor P_3. The four regions packed are illustrated in figure 5.11 (c.f. 5.11(b)) and are numbered in the order they are sent. The receive order by processors owning static partition regions, however, can be arbitrary. The figure (c.f. 5.11(c)) also shows an overhead view of the mapping to illustrate the guard region interaction.

There are important details to consider related to managing the interprocessor communication scheme. We need to ensure that *the communication is as balanced as possible*, due to the irregularity in messages sent and received among partitions. Additionally, *the message passing protocol must be deadlock-free.* Since the fields transported may potentially be large and since complex, yet efficient, computations are performed in determining how fields are passed among processors, a scheme that overlaps communication with computation is desirable.

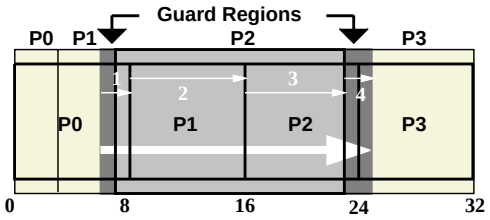
Since every processor knows how many sends to perform and receives to expect,



(a) Scan across moving region on P_2. Guard regions on P_2 are indicated.



(b) Four regions are packed from P_2 and transported resulting from the scan operation.



(c) Overhead view of guard region interaction and block creation.

FIGURE 5.11: Illustration of scan-pack-move method for processor P_2 mapping into the static partition regions.

using blocking sends followed by blocking receives would be straightforward to implement. Unfortunately, this approach can cause deadlock, as seen in figure 5.12 on page 115. Using the scan-pack-move algorithm the following sequence of events may occur:

1. Processor P_1's moving region *sends* a field region to processor P_0's static region.

2. Processor P_1's static region *waits* to receive a message from any processor; it expects a message from the receive-posting algorithm.

3. Processor P_1's static region *can only receive if a message is sent, but the only message that can arrive is from processor P_1's moving region*, due to the scan direction.

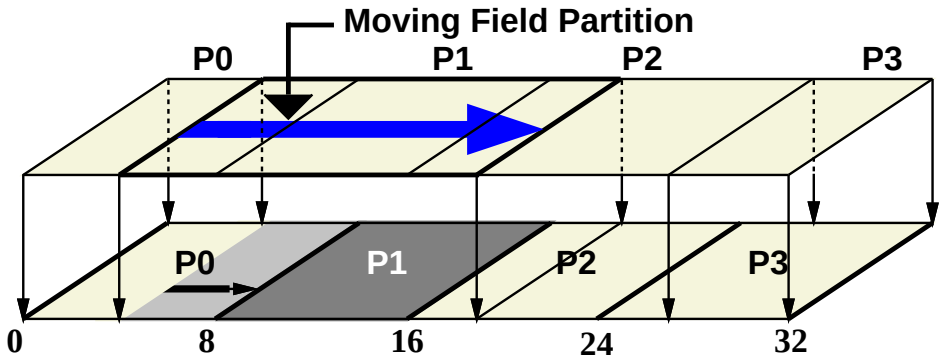


FIGURE 5.12: Deadlock can occur in the scan-pack-move method for processor P_1 if the communication protocol does not allow messages to be sent whenever possible. P_1's static region will wait for a message from P_1's moving region that can never be sent. (P_0's static region successfully receives a portion of the field; P_2's region will not.)

4. The process on P_1 is *blocked forever* waiting to receive a field segment that cannot be sent.

This deadlock occured because the communication protocol waited for a message that was expected rather than continuing with other work. We can eliminate this possibility, and allow concurrent processes to unblock themselves if necessary, by using a protocol that sends fields whenever possible. An advantage of this approach is that communication and computation can be overlapped. For this scheme to work, however, the algorithm must be designed to receive field messages at any moment.

The details of the communication protocol are described in [43], however, figure 5.13 illustrates the major ideas of the approach. While the scan-pack-move algorithm appears to be linear in the number of processors, since we mathematically compute the block sizes rather than actually scanning normal to the partitioning direction, it is really linear in the size of the field [43].

The Inverse Mapping (Step 3) The inverse mapping from the static partitioned regions to the moving regions is similar to the forward mapping. Since an inverse mapping table was constructed the location, size and destination processor of static field

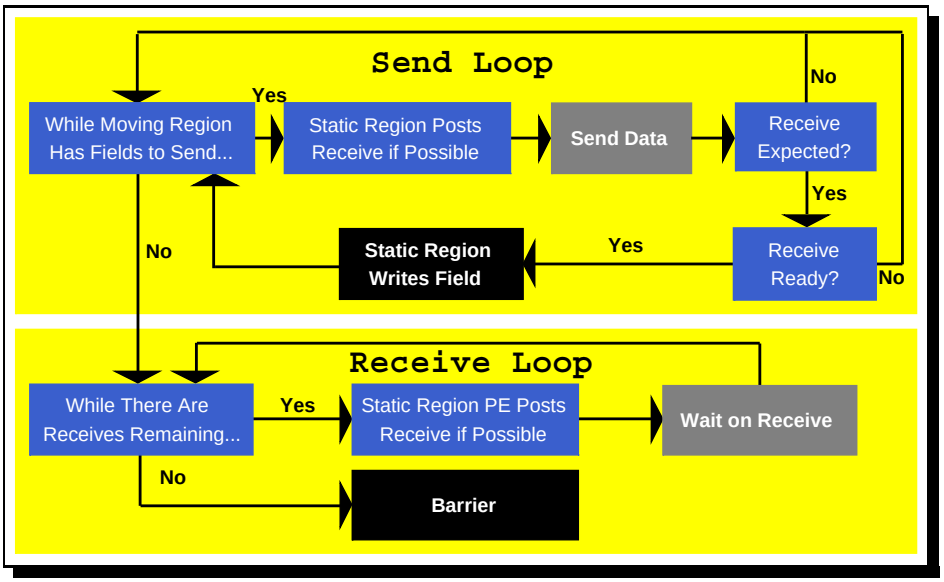


FIGURE 5.13: Illustration of scan-pack-move algorithm with interprocessor communication protocol.

Inverse Scan–Pack–Move Algorithm

1. Read inverse mapping information from table and use to pack and send field components.
2. Continue (step 1) until the inverse table has been exhausted.

FIGURE 5.14: Sketch of the Scan-Pack-Move Method in Inverse Mapping.

components can be accessed directly. In fact, scanning in this case is simply reading the dynamic inverse table and following the instructions, as shown in figure 5.14.

The inverse mapping moves the complex Fourier fields into the real electric vector field, which involves a much larger data transfer than the charge density field. Although the static field regions do not contain guard cells the inverse mapping must account for them in the moving regions. The content of the inverse table data automatically accounted for this. This section of the algorithm is bounded by the size of the static partitioned field on a single processor, but the proportionality constant is larger since a vector field is transported.

Additional Remarks Control in the scanning partition algorithm is *decentralized*; no single processor is responsible for the execution of the algorithm. Additionally, there is *no explicit synchronization* during the algorithm, although a barrier is established at the end of the algorithm to ensure that all field components have arrived at the appropriate processor before the Fourier transforms. This is necessary since the FFT uses matrix transposes on the statically partitioned fields where message passing is required. The overall complexity of the scanning partition algorithm is minimal, linear in the number of field elements, as expected since particles are not involved.

5.3 Load Balancing in the Plasma PIC Algorithm

Many applications programmed on distributed memory parallel machines require load balancing; particle computations are no exception [12, 32, 35]. For our purposes, since the number of particles is much larger than the number of field elements, the particles must be reasonably well-balanced across processors to maintain efficiency. The particle motion influences the scalar and vector field partitioning, since particles must interpolate charge and force data to/from the fields locally. Balancing the particles for computational efficiency is the source of irregularity in the field and particle decomposition.

The plasma PIC algorithm shown in figure 2.2 on page 16 can be modified for load balancing. Our approach, developed independently, is very similar to that described in [12] where an approximation of the plasma density profile is computed to determine the new partitioning. We must know the density profile across the distributed field because, for some simulations, the imbalance evolves over time requiring dynamic repartitioning. We can compute the profile in a simple way by examining the particle interpolation to the grid, using a number density function computed on each processor. These values can be globally combined to find the density profile across the distributed field, allowing a new partitioning to be determined. The location of the

partition borders should balance the number of particles, within reason, given that we have resolved the borders to integral field addresses for simplicity.

The number density function can be computed directly from the existing charge on the field by weighting the density array by a specific value, say the inverse of the species charge. For simplicity and experimental purposes however, we computed the exact number density from interpolation of particles to their near-neighbor grid point in the plane normal to the partitioning direction, for three-dimensions. This has the disadvantage that computing the density now depends on the number of particles, but it is simple and direct. Furthermore, *our interest is not to improve on an existing load balancing algorithm, but rather to investigate how object-oriented methods can be useful in load balancing.* The mechanism described can be changed to utilize the existing charge, when convenient. Fortunately, the encapsulation of simulation components allows for such modifications without affecting the program organization.

We can establish a perfect balance factor $\beta_p = \lfloor \mathrm{N}_p/\mathrm{N}_{proc} \rfloor$ where N_p is the total number of particles and N_{proc} is the number of processors. The balance limits, which indicate the perfect number of particles "to the left" of processor k's left and right partition are:

$$\beta_{lp}^{(k)} = \mathrm{PE}_k * \beta_p \tag{5.2}$$

$$\beta_{rp}^{(k)} = \mathrm{PE}_{k+1} * \beta_p \tag{5.3}$$

where $(\beta_{lp}^{(k)},\ \beta_{rp}^{(k)})$ are the limits on the number of particles with coordinates less than the left and right partitions respectively and PE_k is processor number k. Then, the location of the left and right borders $(\mathrm{PE}_L,\ \mathrm{PE}_R)$ on a specific processor are

determined from:

$$\mathrm{PE}_k * \beta_p \leq \int_0^{\mathrm{PE}_L^{(k)}} n(z)\, dz \tag{5.4}$$

$$\mathrm{PE}_{k+1} * \beta_p \leq \int_0^{\mathrm{PE}_R^{(k)}} n(z)\, dz \tag{5.5}$$

where $n(z)$ is the particle number density function normal to the z slab partition direction. (The absolute left and right borders are fixed at zero and the system length in z, respectively.) This implies that we can count, up to the balance limits, the running sum of the number density to find the new left and right partition addresses. Equations 5.4 and 5.5 can be solved for $\mathrm{PE}_L^{(k)}$ and $\mathrm{PE}_R^{(k)}$ by finding the global index z in the number density array that satisfies:

$$\mathrm{PE}_L^{(k)} = \max\, z \text{ s.t.} \sum_{z=0}^{\alpha_R} n(z) \leq \beta_{lp}^{(k)} \tag{5.6}$$

$$\mathrm{PE}_R^{(k)} = \max\, z \text{ s.t.} \sum_{z=0}^{\alpha_R} n(z) \leq \beta_{rp}^{(k)} \tag{5.7}$$

where z corresponds to the global partition address determined simultaneously on each processor, and α_R is the absolute-right simulation border. *The computation of the running sum implies that a processor could be forced into choosing a left border which exceeds the balanced average, possibly by a significant amount, due to the integral mapping of partition addresses.* The selection of the right border tends to cancel this out in general, although the final processor could contain fewer particles than its predecessors. We have also introduced a scheme that uses recursive bisection of the particle number density across the field, but selection of integral field addresses can introduce the same difficulties recently described.

An important issue involves when load balancing should be detected and performed. Since our scheme involves examining the particles to compute the number density function, there are complications in this decision. One option is to rebalance

after the `Advance` routine (c.f. figure 2.2 on page 16). This has the benefit that since particles will be moved to new processors in the next step, any additional particles that must be moved due to changes in partitioning will also be handled. The major problem is that we do not have an accurate count of the number of particles per processor after `Advance`, so we do not know if load balancing is necessary. Realistically, the severity of this issue depends on the simulation since it may be fine to delay repartitioning until the next time step.

Another option is to load balance after the particles are moved across processors, since the number per processor is now known exactly. The disadvantage with this scheme is that when load balancing is required particles must be moved across processors again, reintroducing interprocessor communication. This can be minimized over the previous approach however, since we have an accurate particle count. This technique will be applied as the scanning partition algorithm is applied for irregular field mappings in load balancing using object-oriented implicit monitoring.

5.4 Continuous Implicit Monitoring for Irregular Computation

We have studied how object-oriented technology is useful in application design for scientific programming. Many applications require load balancing, which is often difficult to introduce into the programming process. In this section, we develop the benefits of abstraction modeling and encapsulation into a scheme useful for load balancing in irregular computation.

The plasma computation is typical of many simulation codes in that modifications can allow new phenomena to be studied. For this application, such modifications may introduce the need for load balancing to maintain efficiency. Oftentimes, programs are instrumented in an explicit and inconsistent manner to measure variables that affect performance. While effective, such a technique could turn a well organized

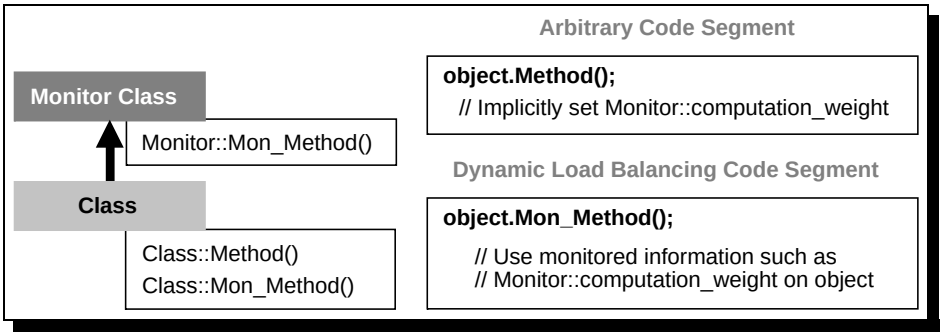


FIGURE 5.15: Implicit monitoring and acquisition of data for load balancing.

program into one which becomes increasingly difficult to maintain as it is modified and features for load balancing are introduced.

Our new approach is an object-oriented instrumentation technique that monitors program abstractions in a *continuous* and *implicit* manner. We will not disrupt a well organized class hierarchy, in fact the method utilizes the features that the object-oriented methodology provides. Objects that must be load balanced are monitored implicitly via normal object usage. The implementation relies on polymorphic operations that allow a wide variety of information to be collected and utilized in unique ways.

Methodology for Object-Oriented Load Balancing The continuous monitoring methodology allows object usage to implicitly gather data useful for run-time load balancing algorithms. The mechanism uses a `monitor class` that is inherited into the application class hierarchy, augmenting derived class object capabilities. The monitor class provides polymorphic operations allowing derived class objects to respond to monitoring messages based on their type. Monitoring then, can be applied to specific objects or the entire collection of monitored objects. Normal application programming is distinct from the activities of the monitor, yet the information collected is easily available to dynamic load balancing routines. Figure 5.15 shows how

an arbitrary code segment calls one of its member functions that also modifies a monitor variable. When that object is used in a dynamic load balancing routine, the information monitored on the object is available for use in balancing operations.

Figure 5.16 shows how the class hierarchy of figure 3.8 on page 56 is extended for continuous implicit monitoring. Since particle distributions are the source of load imbalance, the species and the collective plasma classes inherit from the monitor class. Now, when instrumented methods are called on objects of these classes, they will be implicitly monitored for use by the load balancing algorithm.

Instrumentation has been used to measure the performance of software applications [26, 27], however object-oriented methodology has not been applied to enhance its effectiveness for usage in load balancing until now. When a derived class inherits from the monitor class, the base class monitor operations can be used to instrument derived class methods to collect information. When a derived class object calls one of its instrumented methods, the code segment will implicitly set monitor data. This is implicit because the calling sequence and use of the derived object throughout the program remain unchanged. Since access routines for monitoring information are also inherited, load balancing code segments can interpret monitored data as required. Virtual functions are used for polymorphism in C++, therefore derived classes must provide definitions for base class monitoring operations. This allows derived classes to provide application specific definitions for generic monitoring routine calls. Examples of this technique will be illustrated for load balancing in the free expansion and gravitational problems.

Collisionless Free Expansion Experiment The collisionless free expansion experiment, described in Chapter 2, models the expansion of electrons and ions (assuming charge neutrality) from the fields created by the respective species charges. Mobile ions are easily added to the simulation program from the species class, supporting this experiment. As seen in Figure 2.4 on page 25, initially the electrons and ions

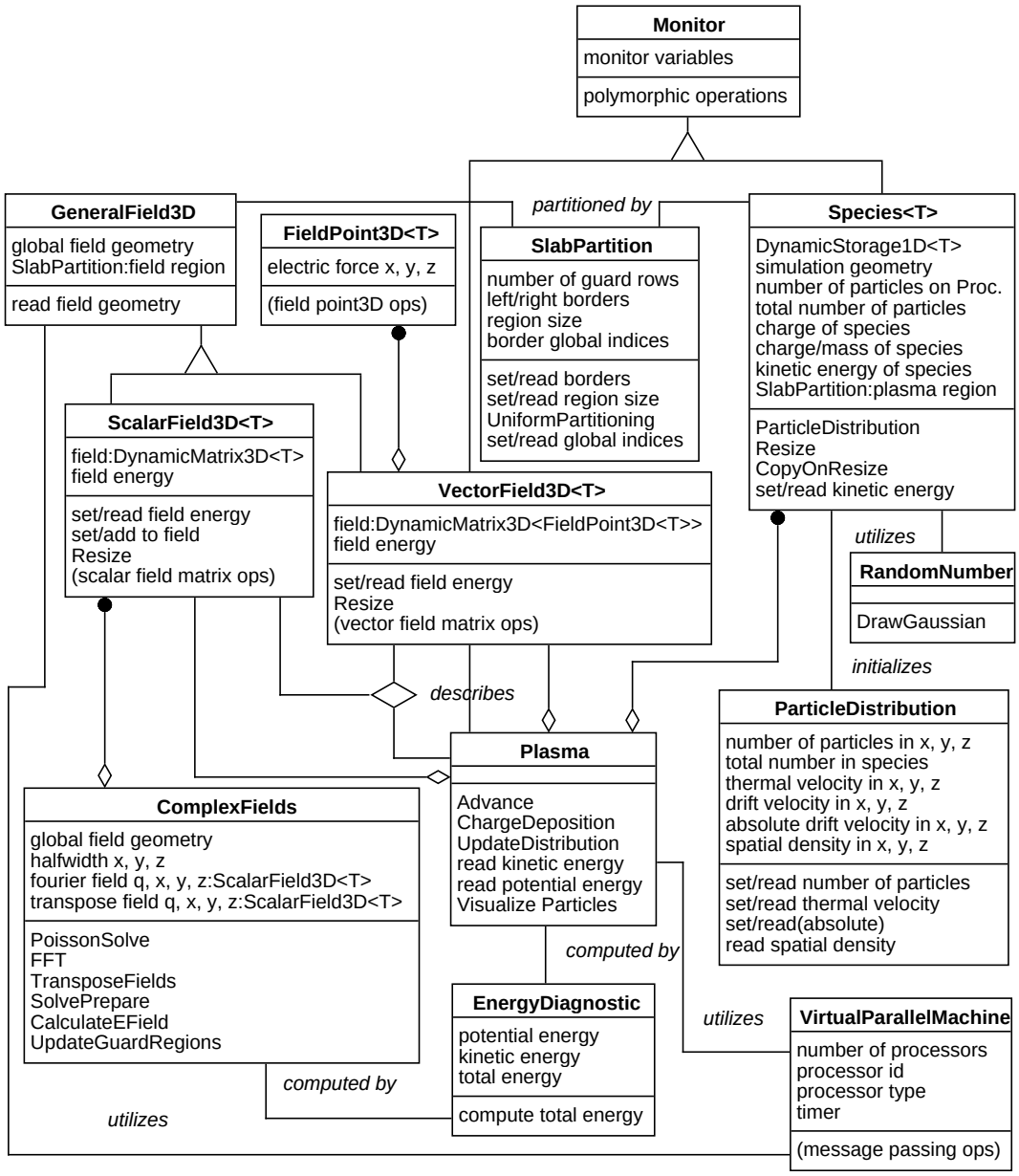


FIGURE 5.16: C++ three-dimensional parallel hierarchy with implicit monitoring class mechanism applied to `Plasma` and `Species` classes.

Object-based Instrumentation of Program Segment

```
void Plasma::UpdateDistribution( Species< Particle3D >& spec,
                                 const VPMachine& vm )
{
  // Code Omitted...
     spec.mon_compwgt( float(spec.npp) );   // monitor species...
     mon_compwgt( spec.mon_compwgt() );     // monitor plasma...
  // Code Omitted...
}
```

FIGURE 5.17: A plasma class code segment is instrumented to monitor the movement of electron and ion particle species across processors. The number of particles, for each species in the current partition, is monitored automatically based on the species type. Additionally, the plasma object monitors the species for usage in load balancing routines. Calling the `UpdateDistribution` routine on the plasma object collects instrumentation data implicitly.

are evenly distributed in the center of the simulation region (c.f. 2.4(a)). Dynamic repartitioning balances the electrons and ions across processors as they move from the dense particle region into the vacuum region. A variety of boundary conditions can be applied including reflecting, emitting, absorbing, or combinations, which may allow for thermal reservoirs of particles to remain beyond the simulation region boundary [10]. In this particular experiment, due to the periodic boundaries, we simply halted the simulation when particles reached the outermost boundaries.

The movement of electron and ion species are the source of irregularity, therefore, these collections were monitored using the species class. This involved calling a monitor generic function (`mon_compwgt`) to set the number of particles for each species per processor. Since electrons and ions are stored on each processor, and since they must be used in computing new partitions, a monitor was added to the plasma class to track the entire collection. The `UpdateDistribution` routine contains the instrumentation, as seen in figure 5.17.

When the main program calls the `UpdateDistribution` routine on the plasma ob-

ject for a specific species, *the number of particles monitored for that object is adjusted implicitly,*

```
plasma.UpdateDistribution( electrons, vpm ); // monitor objects
plasma.UpdateDistribution( ions, vpm );      // implicitly...
```

where the information monitored on an object can now be accessed within load balancing routines via monitor class member functions.

Virtual functions[1] are applied to the species and to the plasma class objects. Since `UpdateDistribution` is called once for the electron species, and again for the ion species, only one code segment for the `mon_compwgt` call is needed, but it will be applied to the proper species. Furthermore, since the plasma class object owns `UpdateDistribution`, it calls a new virtual definition of the `mon_compwgt` routine. This version computes the running sum for all species moved across partitions when multiple species contribute to irregular particle distributions. By monitoring the specific species, and the collection of plasma particles, this information can be used by the load balancing code segment to determine if the imbalance is sufficient to require recomputation of new partition borders, as seen in figure 5.18. The load balancing code internally refers to monitor routines based on the objects it needs to balance. The implicit monitoring method encapsulates the features of load balancing while providing a technique to instrument and operate upon objects targeted for redistribution.

Figure 5.19 on page 127 shows the movement of partition borders for a small expansion experiment using four processors. The initial balanced partitioning is modified as the plasma expands. The time steps when load balancing occurs are indicated with the changes in partition borders. Table 5.2 on page 128 shows the initial and final partition border locations for the processors where particle domains

[1] The `->` operator is typically used to invoke a virtual function in C++. Since a reference argument is passed to the routine, the "." operator is applied instead. Also, since the plasma object is monitored from within its own member function, the `this*` pointer automatically invokes the virtual monitor function.

Usage of Monitored Information in Load Balancing

```
// Load balancing call from main program...
plasma.BalanceDistributionRB( electrons, ions,
                              cdensity, efield, vpm );

// Access monitored data on plasma object within call...
is_balanced_required = f( mon_compwgt(), ... );
mon_init();                   // reinitialize monitor on object...
```

FIGURE 5.18: Load balancing operations can be called on the plasma object. Within the routine, the monitored information is automatically available, where the data has been collected implicitly by other object operations.

are shaded. Although the plasma expands uniformly requiring load balancing, there is an extended period after which repartitioning is no longer required. Nevertheless, the final partitioning could be improved if additional computations were applied in evaluating the quality of the partitioning. This has also affected the appearance of the intermediate partitioning in figure 5.19.

The monitoring mechanism can easily be extended to retain information collected, useful in examining trends in object distribution. Such an extension allows for new partitions to be created, or new load balancing conditions to be applied, based on preferential movement of objects in the simulation. This was not necessary for our purposes since particles are the only objects to balance, but the methodology does allow for such extensions. Figure A.15 on page 211 shows the main program for this experiment.

Gravitation Experiment Although implicit monitoring using object-oriented techniques allowed for load balancing in the free expansion experiment, the nature of the expansion did not introduce a high degree of imbalance. The plasma PIC code can be modified for experiments in gravitation, as described in Chapter 2, where high and low density particle regions are easily introduced from clustering.

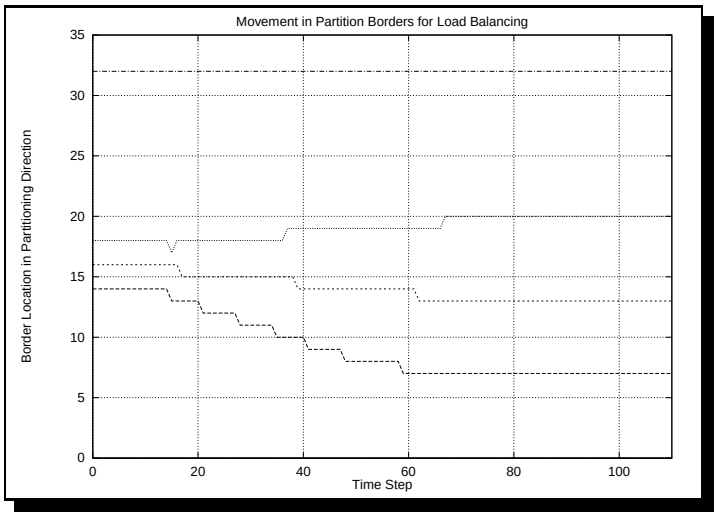
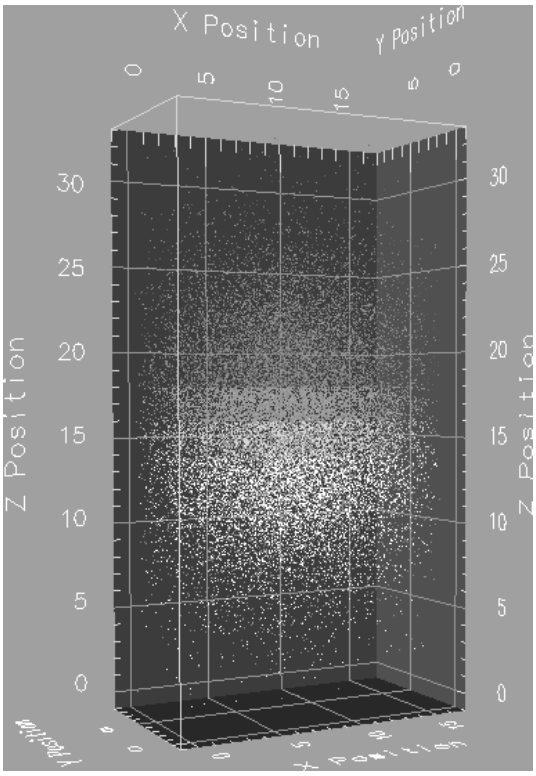


FIGURE 5.19: Movement in partition borders for free-expansion experiment using four processors and the implicit monitoring technique. *The particle expansion is symmetric, however, the partitioning scheme is simplistic so symmetry in border movement is not maintained.*

In this experiment, the motion of a single particle species will be followed. A cubic geometry with triply periodic boundary conditions is applied so that the average mass density within the simulation region remains constant [25]. Figure 5.20 shows the change in partition borders during the gravitational experiment, indicating the creation and movement of high density regions during the experiment.

In figure 2.5 on page 26 the initial distribution of bodies, projected against the partition direction, was illustrated (c.f. 2.5(a)), as well as a three-dimensional view of the simulation space during execution (c.f. 2.5(b)). Figure 5.21 on page 130 shows a portion of the evolution of the system, and the partitioning for one time step, when load balancing is applied using the implicit monitoring technique. The approach is identical to that of the free expansion experiment, however, only the electron species is monitored. Table 5.3 on page 129 shows the partitioning addresses. The ability to observe specific objects in this manner simplified the load balancing issues associated with modifying the code for this new experiment. Appendix A.16 on page 212 shows the main program for the gravitational problem, although the major changes were



(a) Final partitioning for free expansion.

Processor ID	Initial Position		Final Position	
	Left	Right	Left	Right
P0	0	14	0	7
P1	14	16	7	13
P2	16	18	13	20
P3	18	32	20	32

(b) Partition border locations.

TABLE 5.2: Initial and final locations of moving partition borders when load balancing is applied in the free expansion experiment.

limited to modifying system parameters.

Concluding Remarks Regarding the dynamic load balancing algorithm that uses data from the continuous implicit monitoring technique, a slab-partitioning bisection method was applied. Although the monitoring was continuous, the bisection method examined monitored objects to determine if repartitioning was necessary. When the variance—the monitored computation weight representing the number of particles per processor—differed from an empirically determined amount, load balancing was applied. If the difference between the optimal number of particles per processor

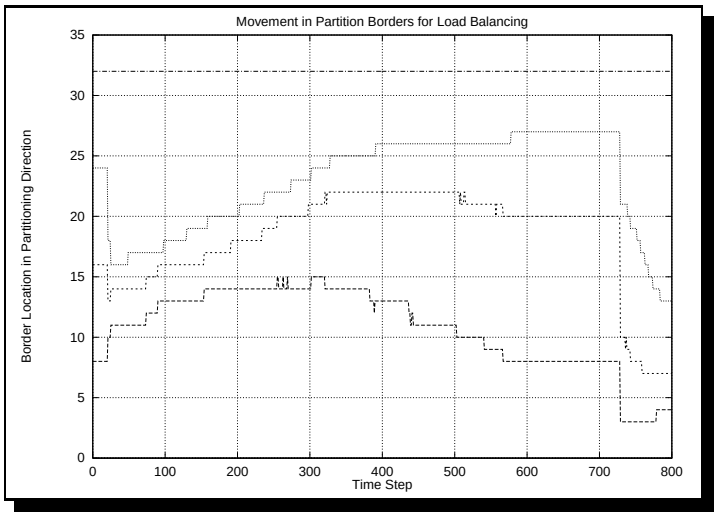


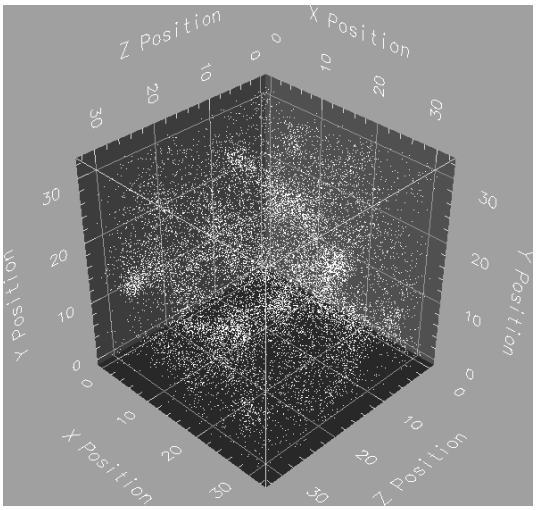
FIGURE 5.20: Movement in partition borders for gravitational experiment using four processors and the implicit monitoring technique.

Processor ID	Initial Position		New Position	
	Left	Right	Left	Right
P0	0	8	0	11
P1	8	16	11	14
P2	16	24	14	16
P3	24	32	16	32

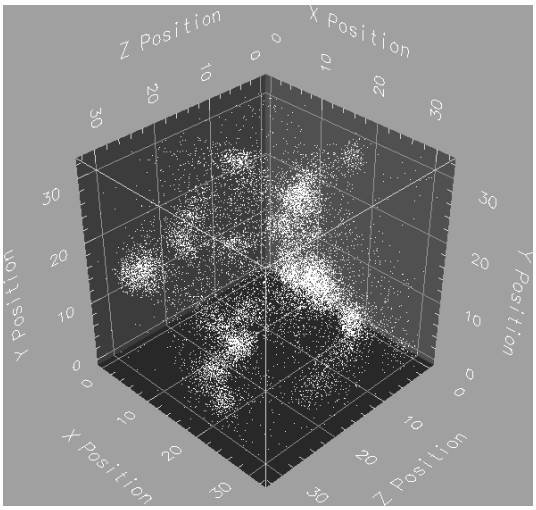
TABLE 5.3: Moving partition border locations at the initial time step and step 25.

and the amount dynamically measured was not significant, control was immediately returned from the load balancing segment to the main program.

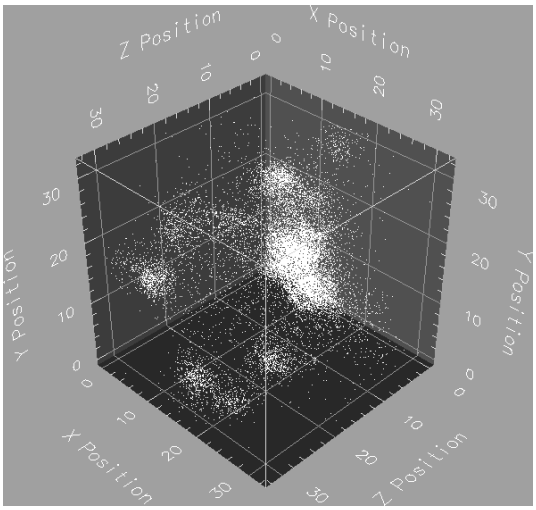
The bisection approach introduced is actually a fast approximation to recursive bisection methods commonly used. Since each processor has all of the particle density information, new borders can be computed based on the processor number and the number of particles expected for a perfectly balanced load. This approximation is sufficient, since the partitions are resolved to integral field addresses and perfect balancing cannot be achieved. Distributions that are nearly balanced tend to be good enough. Moving partition borders can be mapped to real numbers rather than



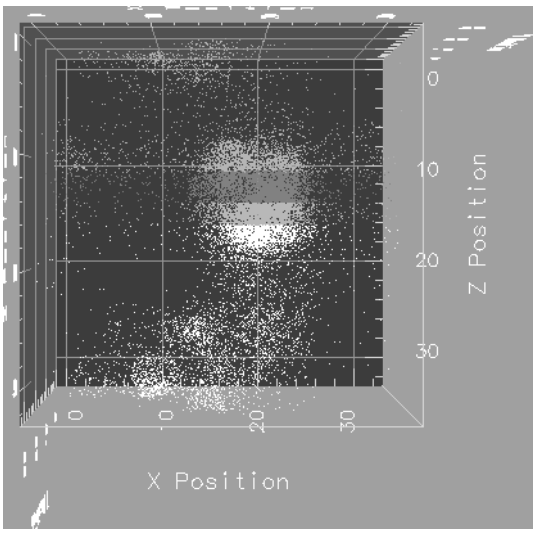
(a) Body distribution for gravitational problem at time step 15.



(b) Body distribution at time step 20.



(c) Body distribution for gravitational problem at time step 25.



(d) Body distribution at time step 25 (overhead view).

FIGURE 5.21: Intermediate development of gravitational experiment when dynamic load balancing is applied.

integers, however, the integer scheme was applied since we were concerned with object-oriented monitoring, not improving upon partitioning schemes. Our choice of the balancing threshold that triggers repartitioning was 15% above or below the known value for a perfect balance across processors. The performance of these programs will be discussed in Chapter 7.

Chapter 6

The Implications of Abstraction

The chapter explores how abstraction modeling can be influenced by programming language features. This affects how programs must be organized at a high level and illustrates how language selection impacts design decisions. In some instances, language constructs open new alternatives useful in abstraction modeling of scientific programs, while in others, these alternatives impose design reorganization.

6.1 The Influence of Language Statements on Object-Oriented Modeling and Programming

Object-oriented methodology is gaining attention as an approach to model abstract concepts and relationships in scientific programming. However, particular language features can influence the ability to design and program with objects. Rather than providing support to represent your ideas, in many instances the language may restrict how concepts can be modeled. Indeed, language features may inversely affect how your program is written instead of providing a mechanism to clearly represent your ideas.

Object technology in scientific programming is appealing since we gain abstraction, application-oriented views, flexibility in design modification, readability, and

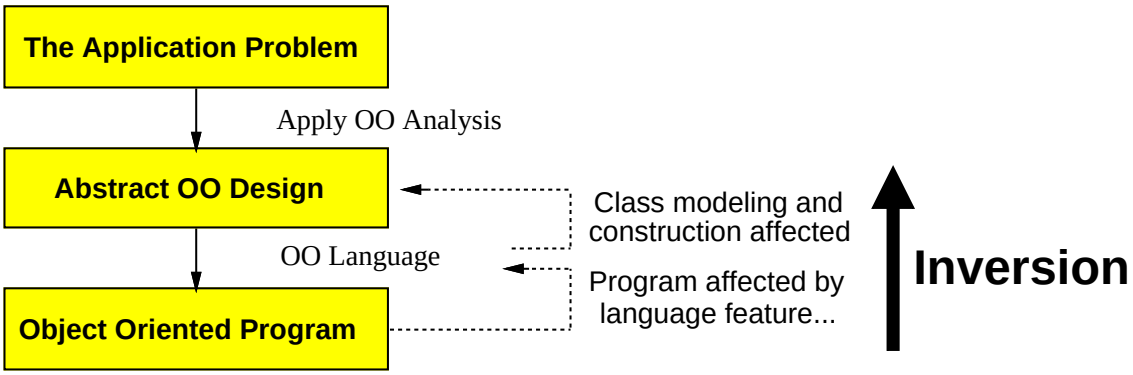


FIGURE 6.1: Inversion in programming, indicating how language statements can cause the redesign of abstraction models.

enhanced collaboration. Valid reasons not to consider object-oriented programming, however, include the difficulty of problem modeling, uncertainty in compiler technology, lack of design standards, the large learning investment, and frequently, performance. The question remains, where does *programming* fit in? Do we ever consider how well a language supports the object model and methodology? Do language properties affect how we model problems in an object-oriented way?

Figure 6.1 illustrates the typical design process for an object-oriented program. Given an application, analysis techniques are employed leading to an object-oriented design. This design, represented by classes, is then implemented in a programming language. During implementation, however, the construction of the program may be influenced by a language specific feature which may prevent representing the abstract design as expected. This affects the class model design, causing an *inversion*—the design of the abstract model must be modified based on a language specific feature. Inversion often causes programmers to build software based not on their modeling abstractions, but on the features a specific programming language provides which reasonably approximates these abstractions.

Inversion in object modeling may not necessarily be harmful; at times it can be beneficial. This is particularly true in scientific programming where abstractions must maintain simulation and physical viewpoints. These modeling issues are important

since they affect reuse, influence programming decisions, determine software understandability, and most significantly, may impose the choice of language. Scientific programming does not represent the origin of object-oriented technology. The modeling techniques applicable to the "real-world" applications driving the creation and development of this technology may apply differently to scientific applications. This issue is examined in C++ and Fortran 90 programming for modeling various plasma PIC abstractions.

6.2 Comparing C++ and Fortran 90 Models of Abstraction

The design goals of the object-oriented PIC model were discussed in previous chapters. We can compare and evaluate the implications of abstraction by examining how C++ and Fortran 90 features affected the organization of basic program segments.

Field Modeling A Fortran 90 one-dimensional `fields_module` segment, seen in figure 6.2, shows how the scalar charge density and scalar electric field derived type definitions are included with the numerical routines that manipulate these fields. This module organization simplifies extension to higher dimensional problems, or mixed-dimensional codes, since additional derived types can be included preserving the original context, as seen in figure 6.3 on page 136, where `vfield2d` defines a two-dimensional vector field.

Fortran 90 statements allow routines to be defined together with *any number* of related objects, yet these objects remain independent. This is because objects in Fortran 90 are arguments to routines made available by use-association. This represents physical and computational abstractions well since separate objects, `cdensity` and `efield`, can be modified individually (as in `fields_fftrx`) or collectively (as in `fields_pois`) by encapsulated module routines. The module routines are not bound

Sketch of Fortran 90 Scalar Field Module

```
MODULE fields_module
IMPLICIT NONE
TYPE sfields1d                              ! field definition...
   INTEGER :: nx                            ! field dimension
   REAL :: wp                               ! field energy
   REAL, DIMENSION (:), POINTER :: p        ! field elements
END TYPE sfields1d
CONTAINS
   SUBROUTINE fields_pois(cdensity, efield)
     TYPE (sfields1d), INTENT (in) :: cdensity
     TYPE (sfields1d), INTENT (out) :: efield
     ! poisson solver routine
   END SUBROUTINE fields_pois
   SUBROUTINE fields_fftrx(fields)
     TYPE (sfields1d), INTENT (inout) :: fields
     ! fft solver routine
   END SUBROUTINE fields_fftrx
END MODULE fields_module
```

FIGURE 6.2: Fortran 90 design of a module for scalar fields. Both the charge density and electric field objects can be created, where the definition of the object types are included with the numerical routines that manipulate them.

to a single object and objects of different types can be created and used from the `fields_module`.

In C++, however, if the field solver routines are defined with the scalar and vector field objects, we might prefer to encapsulate this into a single class:

```
class Field {
  private:
     float cdensity[ DIM_X ][ DIM_Y ];              // scalar field
     Vector2D< float > efield[ DIM_X ][ DIM_Y ];    // vector field
  public:
     void FFT2D();         // Routines have direct access
     void Poisson2D();     // to all field components...
};
```

where `Vector2D<T>` defines the components of a two-dimensional vector field point.

Fortran 90 Module Extended for Vector Fields

```
MODULE fields_module
IMPLICIT NONE
! sfields2d derived type definition...
TYPE point2d
   REAL :: x, y
END TYPE point2d
TYPE vfields2d          ! 2d-vector field derived type...
   INTEGER :: nx, ny
   REAL :: wp
   TYPE (point2d), DIMENSION (:,:), POINTER :: p
END TYPE vfields2d
CONTAINS
   SUBROUTINE fields_pois(cdensity, efield)
     TYPE (sfields2d), INTENT (in) :: cdensity
     TYPE (vfields2d), INTENT (out) :: efield
     ! poisson solver routine
   END SUBROUTINE fields_pois
END MODULE fields_module
```

FIGURE 6.3: Fortran 90 design of a module for two-dimensional scalar and vector fields extended from the one-dimensional version. New field derived types and related functions can be added, preserving the structure of the original one-dimensional definition.

Then, all operations are bound to *one object*. This approach restricts the accessibility and hides the visibility of the fields since they are deeply encapsulated into the field class. The implications of this organization are that all operations on the charge density and electric fields must be performed on behalf of the single field class object. While this design has one of the benefits of the Fortran 90 version, the fields are immediately accessible to the appropriate numerical routines, they cannot be manipulated as independent objects.

Since it is preferable to manipulate the fields individually, while supporting field solver operations collectively, inversion causes a reorganization of the C++ design. A new class has been added (`Fields2D`) supporting field interactions, as seen in figure 6.4, where *many* supporting details have been omitted. (The dynamically al-

C++ Structural Design for Separate Fields

```
template <class T> class ScalarField2D : public GeneralField2D {
  private:
     DynamicMatrix2D< T > *p;           // scalar field
  public:
     float wp;                          // field energy
};
template <class T> class VectorField2D : public GeneralField2D {
  private:
     DynamicMatrix2D< point2D< T > > *p;  // vector field
  public:
     float wp;                            // field energy
};
class Fields2D {
  public:
     void Poisson2D( const ScalarField2D< complex >* cdensity,
                           VectorField2D< complex >* efield );
     void FFT2D( GeneralField2D< complex >* fields );
};
```

FIGURE 6.4: C++ design of classes supporting the definition and usage of individual scalar and vector fields in numerical field operations. C++ templates allow fields to be resized dynamically while supporting Fortran 90-style array operations.

located fields support Fortran 90-style array operations.) Now, charge density and electric fields can be manipulated individually, and made available to numerical routines, at the high level of the main program rather than the low level of the field class definition.

If a mixed-dimensional code was of interest we might *try* to create a single module, which simply "collects" definitions from previously defined modules, where routine argument types distinguish which function will be called. An example is illustrated in figure 6.5. While this kind of member function overloading is very natural in C++, Fortran 90 requires a different organization. The routine names must be distinct in the implementation, but a Fortran 90 *module procedure* can be introduced to perform the overloading, as seen in figure 6.6 on page 139.

Fortran 90 Incorrect Design for Multiple Field Solvers

```
MODULE wrong_fields_module
IMPLICIT NONE
! sfields1d derived type definitions...
! sfields2d derived type definitions...
! vfields2d derived type definitions...
CONTAINS
   SUBROUTINE fields_pois(cdensity, efield)
     TYPE (sfields1d), INTENT (in) :: cdensity
     TYPE (sfields1d), INTENT (out) :: efield
     ! 1d poisson solver routine
   END SUBROUTINE fields_pois
   SUBROUTINE fields_pois(cdensity, efield)    ! Error, incorrect
     TYPE (sfields2d), INTENT (in) :: cdensity ! overloading...
     TYPE (vfields2d), INTENT (out) :: efield
     ! 2d poisson solver routine
   END SUBROUTINE fields_pois
END MODULE wrong_fields_module
```

FIGURE 6.5: Incorrect Fortran 90 attempt to unify multiple field solvers and definitions into a single abstraction. Function overloading requires Fortran 90 module procedures with user-specified resolution of routine name conflicts.

This allows multiple field solvers with multiple field objects to be defined, collectively, in a module. In this case, creating the field module abstraction requires resolution of operation names by the programmer by use of a Fortran 90 interface block. Resolution of overloaded names is handled automatically in C++.

Particle and Collective Abstractions Field modeling is just one example; particle abstractions can also be affected by language features. We have seen that describing particles as vectors had advantages in preserving mathematical operations across simulation dimensions. Position and velocity components were represented individually, however, since they must be modified independently when vectors are used. Since there was no mechanism to operate on only a single component (position or velocity) of the vector representation using vector operations, a reorganization was

Fortran 90 Correct Design for Multiple Field Solvers

```
MODULE fields_module
IMPLICIT NONE
PRIVATE :: sfields_pois, vfields_pois
INTERFACE fields_pois
   MODULE PROCEDURE sfields_pois
   MODULE PROCEDURE vfields_pois
END INTEFACE
CONTAINS
   SUBROUTINE sfields_pois(cdensity, efield)  ! details
   SUBROUTINE vfields_pois(cdensity, efield)  ! omitted...
END MODULE fields_module
```

FIGURE 6.6: Correct Fortran 90 design unifying multiple field solvers and definitions into a single abstraction. Scalar and vector fields can be solved by calling the `fields_pois` routine, where the correct function is called based on the field type.

induced leading to the species classes already described.

Additionally, Fortran 90 modules and the `use` statement encourage unification of concepts. Building a `plasma_module` with mixed species and field operations is straightforward,

```
MODULE plasma_module
  USE species_module
  USE fields_module
CONTAINS
  SUBROUTINE plasma_dpost (species,sfields)
  ...
  SUBROUTINE plasma_push (species,fields,dt)
  ...
END MODULE plasma_module
```

where the use statement makes the features of the species and fields modules available to routines in the plasma module. Multiple inheritance might be applied to form a similar model in C++, where the features of existing classes are combined into a new class. Unfortunately, this would not be an accurate representation of the abstraction

since the plasma class should not be viewed as a species and a field by an IS-A relationship, only as the unification (collection) of these abstractions. In this case, the reorganization introduces a `plasma class` which only contains functions and no data.

```
class Plasma {
  public:
     void ChargeDeposition( const Species< Particle3D >& spec,
                            ScalarField3D< float >& cdensity );
     void Advance( Species< Particle3D >& spec,
                   const VectorField3D< float >& efield,
                   const float DT );
};
```

These *organizational abstractions* can be very useful in scientific programming. Designers should not feel that every class must contain data.

Performance Implications of Abstraction The design of abstractions affects performance, in particular, objects need to be designed so that the abstractions do not adversely affect efficiency. While it is not possible to generalize how specific abstractions affect performance, we can illustrate when this has occured from our experience. This represents an additional instance where program modeling was affected by language features. (Formal performance measurements are covered in Chapter 7, which include measurements for various Fortran 90 constructs used in an object-oriented manner.)

Our original Fortran 90 programs used allocatable arrays, which allows abstractions, like the fields, to be created based on simulation parameters at run-time. To build better field abstractions we wanted to use dynamic arrays within derived types. Since allocatable arrays are not permitted as components of derived types, pointers to arrays were introduced instead. Pointers can be efficient in Fortran 90 when the `target` attribute specifies variables pointers can reference. Additionally, the use of Fortran 90 pointers includes restrictions to improve efficiency [9]. Unfortunately, pro-

grams containing pointers to arrays apparently execute more slowly than those using allocatable arrays. The compilers are very new however, so this could be a temporary problem. Nevertheless, defining field and species elements within derived types preserves important abstractions; currently the performance difference is tolerable given these advantages.

Fortran 90 supports multidimensional dynamic data structures, which use array syntax operations. Compilers can remove some performance problems through optimization of array loops after the initial intermediate code is generated [37]. Sometimes the optimizations are very beneficial and other times they can still induce performance degradation. Array operations are very useful, but they are not part of the C++ language. Various efforts are in progress to include array libraries, such as in the HPC++ project [63]. Usually most programmers develop their own multidimensional dynamic array libraries which are complex to design and make efficient. Our multidimensional array libraries use dynamic one-dimensional arrays with index sets for addressing. Still, we must take care in their usage and implementation. For instance, updating a vector field element often involves propagation of overloaded array operators through the template field components definition classes, down to the template dynamic array definition and back again. While the field abstraction is beneficial, many of these operations could be removed if the compiler generated the array operation code with known optimizations (such as temporary elimination [21]). Some compilers allow the in-lining depth, the number of levels to which in-lining will be applied, to be specified. Unfortunately, only one level of in-lining was usually allowed.

6.3 Comparing Fortran 77, Fortran 90, and C++ Paradigms

Our experience allows for the comparison of applications developed across a variety of languages and programming paradigms. There has always been some interest

in how traditional object-oriented languages compare to each other [33], but this has not been based on comparisons for a single complex scientific application. Now that we have seen examples of these languages used in a variety of ways, we can compare them against each other to assess their impact on the future directions of scientific programming. Some of this commentary will be supported by performance comparisons in Chapter 7.

Implications for Parallel Programming 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 C++ `VirtualParallelMachine` class, support parallel programming with object methods that transport data using its full object type. The Fortran 90 `MPI Module` provides access to message passing routines by use-association. 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 created, we observed that many parallel programming details can be information hidden within C++ classes and Fortran 90 modules. As HPF becomes more widely available, data parallel programming through distribution directives combined with the object methodology could be an important programming paradigm for this language.

Compiler Stability Issues The efficiency of Fortran programs is commonly cited as the major benefit over C++, yet this may not be as important an issue as compiler stability across machines. As we will see in Chapter 7, this was a major problem in C++ across various architectures. The modifications for abstractions necessarily

caused our Fortran 90 and C++ source programs to be longer (about 2 and 2.5 times respectively, on average) 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 probably much longer than for Fortran 90 since many details must be considered when writing large C++ programs. C++ programs often are slower than Fortran 77 and Fortran 90 programs since optimization across pointer structures is often limited compared to static arrays. This issue must be weighed against the costs of program maintenance, which is a growing concern of many application programmers.

Benefits of Fortran 90 over C++ Fortran 90 has a number of strengths over C++ for scientific computing. These include the built-in array operations, multidimensional dynamic memory management and backward compatibility to Fortran 77 for incremental development. Fortran 90 is also a straightforward extension of Fortran 77, although Fortran 90 is a very different language when the modern features are used. The protection mechanisms, intrinsics, efficiency, object-oriented features, and relation to High Performance Fortran also make Fortran 90 attractive for advanced parallel computations. The Fortran 90 pointer system and usage is very cleverly designed. Many of the standard features in Fortran 90, such as array operations, are being developed as library extensions in C++.

Many other features including intent attributes, optional arguments (which are more general than simple default values), and stronger type checking are additional benefits that Fortran 90 has over C++. The type checking issue can be both a benefit and a weakness since certain features, such as inheritance, had to be constructed explicitly. Nevertheless, strict typing in Fortran 90 leads to a safer program as source code is shared and modified by multiple developers, and it allows the compiler to find even more programming errors.

The most significant benefit of Fortran 90 over C++ is that experienced program-

mers can apply abstraction modeling to their software development without changing to an entirely different language. Many major scientific software projects are based on Fortran 77. While C++ offers the highly publicized benefits of object-oriented modeling, it is unlikely that most Fortran programmers will adopt a new programming style *and* language simultaneously—particularly if this means rewriting extensive programs from scratch.

Advantages of C++ over Fortran 90 C++ is very popular—it is much more well-known than Fortran 90 since the language has been available longer. The template mechanism and typing system are beneficial in abstraction modeling. Additionally, the language can be integrated with modern tools, such as visualization systems. The foreign language interface supports multi-paradigm programming and new additions, such as meta-types and the Standard Template Library, introduce new modeling options that will influence the design of advanced software systems. C++ is a rich language with many features, but it is also very complex and it can be difficult to determine the implications of using advanced techniques. Unfortunately, many C++ compilers have faults, particularly in the template instantiation mechanisms and in data structure memory alignment. The performance is less efficient compared to equivalent Fortran 90 programs and there is a smaller user base in scientific computing compared to Fortran 77.

While templates were used in the advanced C++ programs, this feature currently is not supported in Fortran 90. The particle species, for instance, could be parameterized using C++ templates which is beneficial in adding new species types quickly. In Fortran 90, new species can also be added to the simulation by including a new derived type in the species module. Modeling abstractions without templates did not adversely affect the Fortran 90 design, but usage of this feature was convenient in C++ programming.

Our experience indicates that the efficiency of Fortran 77 and abstraction modeling

capabilities of C++ are desirable features for scientific programming. Converting Fortran 77 programs to Fortran 90 was always faster than converting to C++. A similar experience may occur if C programs were converted to Fortran 90 and C++, but such an experiment was never performed.

Emulation of certain C++ features in Fortran 90, such as sub-typing inheritance, may be too complicated to find general usage. One of the major benefits of C++ over Fortran 90 is the wealth of educational materials and experienced users. Additionally, advanced features including namespaces, expression-templates, and exception handling may find extensive usage in scientific programming as compilers are developed to support these features.

6.4 Commentary

When considering the implications of abstraction an important issue is how well abstractions support program modifications and extensions. Furthermore, how well does a program designed for extension compare to an equivalent version that is not so designed and what implications does this have on abstraction modeling?

When extending the C++ program from the instability experiment to the free-expansion problem moving ions were introduced. In figure A.15 on page 211, the code shows how the species class was used to create the collective ion species where parameters specify the attributes of the objects created. A number of important events occur when this object is created, including storage allocation and species partitioning across processors. Since the C++ species constructor already accounts for these issues, introducing ions into the system only requires adding the ion collection:

```
Species< Particle3D > ions( ICHARGE, ICHGMASS,
                            (backidf.npxyz() + beamidf.npxyz()),
                            SYSLENX, SYSLENY, SYSLENZ, vpm );
```

Including the ion species implies that some routines need to be called multiple times, once for the electrons and again for the ions, such as `Advance` and `Deposit`.

Additionally, more costly routines must also be called on the individual species including `UpdateDistribution`,

```
plasma.UpdateDistribution( electrons, vpm );
plasma.UpdateDistribution( ions, vpm );
```

which requires message passing communication. The electron and ion abstractions only differ in parameterized constants, so they could be organized into a single abstraction, perhaps as charged particles. This reorganization would reduce the number of function calls on objects. Nevertheless, the simplicity of adding the new species, while preserving existing routines, outweighs any possible benefit to designing a new code that may be less flexible. A less noticeable benefit of this approach is that reversion to existing codes simply consists of removing existing statements. The gravitation code was designed by removing the ions from the load balancing free-expansion code, with additional small modifications. This is the greatest benefit and implication of abstraction in scientific programming.

Abstraction modeling can introduce dependencies in object-oriented programming that may or may not be beneficial. In particular, the construction of some objects may depend on the state of other objects in the system based on the simulation performed. In fact, in scientific programming, this may be prevalent.

For example, in the C++ free-expansion experiment the scalar and vector field object partitioning depends on the initial particle distribution. This distribution is based on the species distribution function objects. When the scalar and vector fields are created the distribution objects (`backdf`, `beamdf`) must be included since they have access to the particle distribution:

```
ScalarField3D< float > cdensity( SYSLENX, SYSLENY,
                                 SYSLENZ, vpm, backdf );
VectorField3D< float >   efield( SYSLENX, SYSLENY,
                                 SYSLENZ, vpm, backdf );
```

The distribution objects are passed to the `GeneralField3D` base class constructor which computes the partitioning information for every processor. This information

is passed to the scalar and vector field derived class constructors which dynamically allocate the fields based on the distribution information. This scheme encapsulates field border computations in the field base class constructors with memory allocation delegated to the derived class constructors based on the properties of the distribution objects.

The benefit is that many detailed features can be handled automatically, but codes must be fully documented to make such dependencies clear. Dependencies that are not essential should never be introduced, nor should dependencies that cannot be applied consistently.

Chapter 7

Machine & Compiler Performance Comparisons

This chapter evaluates the performance of the programs developed when compared across various machines, languages, and compilers. The plasma PIC algorithm has been benchmarked thoroughly and is known to be scalable, therefore we address basic performance comparison issues rather than scalability issues.

7.1 Development Experiences Across Compilers & 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, SP series and 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 July 1996. Software problem reports have been filed and are under investigation.

Experiences in Portability The C++ `VirtualParallelMachine` 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 for 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 that machine-specific classes should decrease in importance; yet the ability to perform message-passing on objects should remain valuable. There are efforts underway to create object-oriented versions of MPI. Some of the work focuses on introducing C++ style language bindings [28] with other efforts consider the transmission of objects [58] corresponding to user defined classes. 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 `VirtualParallelMachine` 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++ currently has the reputation of being less efficient. Designing efficient and portable C++ codes is difficult due to differences in compiler implementations. In-lining is touted as "the solution" to the overhead associated with calling methods on objects,

however 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.

The overhead associated with accessing memory also contributes to inefficiency. Our particle representations use object arrays, which require special algorithms to maintain data structure consistency when particles cross processor partitions. This approach allows for larger simulations since arrays use memory more productively than lists for example. 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.

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 routine 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 is then assigned to the associated object using its interface, to preserve encapsulation and protect non-public data. Although direct access to protected data by the message passing routines would violate encapsulation this may be appropriate for efficiency reasons, similar to

usage of the C++ `friend` statement for efficiency. Our original field model used template grid points that maintained 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 for transmission of non-contiguous data, can address this issue. Nevertheless, this illustrates how the abstraction modeling features can influence the efficiency of accessing data and code design since the new field models store charge and force data separately.

Reliability Issues Many useful features for programming abstraction are provided by C++ and Fortran 90; nevertheless, the reliability of existing compilers must be considered. Reliability issues are noticed most clearly during the compilation process. Valid C++ programs which 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. On the other hand, our Fortran 90 scalar programs were ported without change between the IBM and HP Fortran 90 compilers. The parallel Fortran 90 program on the SP2 using the IBM xlf90 compiler was totally reliable. Unfortunately, the Cray Fortran 90 compiler on the T3D currently does not (as of August 1996) support modules which are used extensively in our programs.

The basic features of the C++ compilers we used were generally stable but, as more sophisticated programming techniques were introduced, compiler bugs severely restricted 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.

	Execution Time (seconds)			
	Unoptimized		**Optimized (-O)**	
	Fortran 77	C++	Fortran 77	C++
Total Time	22.00	33.00	3.80	23.00, [**9.0** pow()]
Operations	% total time in function and seconds in function alone			
Advance	24.40% **7.71**	19.10% **8.39**	50.40% **3.41**	21.20% **5.51**
Deposit	13.70% 4.31	14.00% 6.16	34.00% 2.30	14.70% 3.82
FFT	1.10% 0.34	0.90% 0.40	3.80% 0.26	0.80% 0.21
Poisson	0.20% 0.07	0.10% 0.05	0.30% 0.02	0.01% 0.02
mcount	26.90% 8.50		7.40% 0.50	
Exponent		**35.70%** **15.68**		**61.40%** **15.96**
	20,480 Particles and 32,768 Grid Points			

TABLE 7.1: Sequential one-dimensional Sparcstation 10 Performance Comparison of Fortran 77 (SPARCompiler Fortran from SunPro) and C++ (g++ v.2.5.7) using the `gprof` profiling tool. The C++ optimized performance improves dramatically using an optimized `pow` routine.

7.2 Analysis of Scalar Performance

A number of sequential programs were designed which are easy to analyze with performance monitoring tools such as `gprof`. For a one-dimensional code on the Sun Sparcstation 10 for a small experiment, table 7.1 shows some interesting characteristics for the major routines in the plasma PIC codes. All other operations profiled represent timings less than the smallest time indicated in the table.

Sequential one-dimensional Fortran 77 & C++ Comparison The C++ program is based on the model of figure 3.3 on page 36 while the original Fortran 77 program was described in Chapter 2. The total execution time of the unoptimized versions indicates that Fortran 77 is 11 seconds faster than C++, (33% faster than

the total running time of the C++ program). This is true even though the percentage time spent in `Advance`, the most computationally expensive routine which updates particle positions and velocities, is smaller in C++ and the time for a single call is roughly equivalent. This is explained from the overwhelming time used by the C++ exponentiation routine `pow`, even when integer arguments are present. The optimized times show that the `pow` routine prevents much improvement in the C++ program; it finishes in 23 seconds while the Fortran version only requires 3.8 seconds. The Fortran 77 and C++ compilers optimize the loop and array references in `Advance` and `Deposit`, but this is lost in C++ due to exponentiation. When an integer `pow` function is overloaded, the C++ performance improves to 9 seconds.

The C++ FFT and Poisson's Equation solver times compete well with the Fortran 77 version. Since `Advance` and `Deposit` access array elements frequently within loops, the inefficiency of the C++ program compared to Fortran 77 may be due to inadequate loop optimizations.

Sequential two-dimensional Fortran 77, Fortran 90, C & C++ Comparison
Table 7.2 on page 155 shows performance comparisons for a small two-dimensional simulation in Fortran 77, Fortran 90, C, and C++. The C program models the same abstractions as the C++ program, except C structures are used to represent objects and object collections. Additionally all data structures are statically allocated, whereas in C++ dynamic data structures are used. For example, a collection of particles is defined in C as follows:

```
typedef struct particle {
        float pos_x, pos_y, vel_x, vel_y;
} PARTICLE, *PARTICLE_PTR;

PARTICLE elec[ NP ];                    /* Array of Particles  */
```

where the function prototype for routines, such as `Advance`, appear as follows:

```
void Advance( PARTICLE elec[ NP ], EFIELD efield[ X_DIM ][ Y_DIM ],
              float *, int );
```

	OPTIMIZED (-O) Execution Time (seconds)							
	Fortran 77		**Fortran 90**		**C**		**C++**	
	total time without (with) profiling instrumentation							
	27.14	(26.97)	**29.80**	(29.43)	**50.00**	(88.44)	**41.00**	(49.01)
Operations	% total time in function and seconds in function alone							
Advance	41.30%	**11.14**	42.80%	**12.06**	13.70%	**12.08**	29.20%	**14.30**
Deposit	26.80%	7.22	27.40%	8.07	9.80%	8.68	21.10%	10.34
FFT	4.90%	1.33	4.20%	1.23	5.60%	4.93	19.80%	9.69
Poisson	0.50%	0.14	0.70%	0.21	0.20%	0.14	0.60%	0.27
itrunc	25.40%	6.86	21.09%	6.44	7.90%	7.00	14.90%	7.32
mcount					**29.60%**	**26.14**	**8.90%**	**4.34**
Exponent					**16.80%**	**14.89**		
	20,480 Particles and 2,048 Grid Points							

TABLE 7.2: Sequential two-dimensional IBM RS6000 Performance Comparison of Fortran 77 (IBM xlf), Fortran 90 (IBM xlf90), C (IBM xlc) and C++ (IBM xlC) using the `gprof` profiling tool.

The C++ program is based on the model in figure 3.4 on page 39 while the Fortran 90 program is from the model in figure 4.10 on page 92.

To learn more about the effects of the integer optimized `pow()` routine, we computed integer exponentials explicitly in the `Advance` and `Deposit` functions. The execution time per call of `pow()` is too small to measure accurately. The cumulative effect is noticeable since 67,288,239 is the invocation count. The effects of this modification are shown in table 7.3 on page 156. The amount of time per call in `Advance` and `Deposit` has decreased slightly due to this modification (compare to table 7.2 on page 155) since the `pow()` routine is not called.

Comparatively, the C++ versions are very competitive to the Fortran versions, but this efficiency is not seen in the overall execution times. Although there was a significant collection of small class member functions within the C++ programs (not

	Execution Time (seconds)					
	C++ Unoptimized		**C++ Opt (-O)**		**C Opt (-O)**	
	Total time without (with) profiling instrumentation					
	70.00	(112.43)	**38.00**	(43.51)	**43.00**	(49.74)
Operations	% total time in function and seconds in function alone					
Advance	21.00%	**23.56**	26.60%	**11.56**	22.60%	**11.22**
Deposit	15.00%	16.87	20.10%	8.76	15.80%	7.88
FFT	6.70%	7.56	21.40%	9.32	10.60%	5.26
Poisson	0.40%	0.45	0.40%	0.18	0.40%	0.18
itrunc	7.50%	8.46	14.30%	6.24	13.10%	6.54
mcount	**32.2%**	**36.15**	**10.60%**	**4.61**	**14.50%**	**7.23**
Exponent	**0.30%**	**0.31**				
complex fcns					**18.10%**	**8.99**
	20,480 Particles and 2,048 Grid Points					

TABLE 7.3: Sequential two-dimensional IBM RS6000 Performance Comparison of C++ (IBM xlC) and C (IBM xlc) without optimized exponentiation calls in major loops, using the `gprof` profiling tool.

shown), when in-lined by the C++ optimizer they only represented about 3% of the total execution time. As a result, in-lining did not contribute very much to performance improvement when the optimizer was applied. Additionally, the time spent in the FFT routine is significant compared to the Fortran 77 and Fortran 90 programs. The one-dimensional FFT routine used a static array allocated at compile-time. The two-dimensional implementation used indirect-addressing.[1] Storage was dynamically allocated as a one-dimensional pointer array where each element pointed to a dynamically allocated one-dimensional array. This approach allows the standard C++ index operator to be used in addressing dynamically allocated two-dimensional arrays. The

[1] All of the other programs with dynamic fields used a linear-addressing scheme where a pointer to a dynamically allocated one-dimensional array with linear index functions was applied.

FFT performance worsened when the optimizer was applied, so we replaced the dynamic allocation scheme with a static allocation scheme for both the FFT and the Poisson's Equation solver. The performance of the FFT and the other portions of the code remained essentially the same. Apparently the dynamic allocation scheme did not contribute to the performance problems in the FFT.

There was an overhead associated with calling complex multiplications from the C++ `complex` library, yet only 1.43 seconds of the 9.52 seconds spent in the FFT represented the time used within this function. The routine was called 5,823,350 times, however, within the FFT. Since the profiler provided no other significant information on the FFT, the overhead in calling the complex multiplication could account for much of the overall performance problem.

Table 7.2, and table 7.4 on page 158, indicate that the object-oriented Fortran 90 performance is competitive with the Fortran 77 programs, however, this may not always be the case.[2] Finally, the C++ optimized program outperformed the optimized C version even though an integer C exponentiation function was used. Table 7.3 on page 156 shows performance results where exponentiation was removed from the `Advance` and `Deposit` routines, however, the C++ program still had a slight advantage over the C program. The overhead within the C program came from calling routines written to perform complex arithmetic. Cumulatively, the time spent in creation of complex numbers with addition, multiplication, subtraction, and conjugate operations are listed in table 7.3 under the `complex fcns` heading.

This issue of the `pow()` routine and complex library functions might seem belabored, but this brings attention to how library routines may affect performance. When these routines are not implemented with care, the effects can be dramatic, and unknown, unless programs are profiled. It is impossible to guess at causes for poor performance; this issue illustrates the importance of measurements.

[2]Viktor Decyk has performed experiments with Fortran 90 compilers on the RS/6000 58H, Sun Sparcstation20, Silicon Graphics Indy, and Cray C90 where performance has varied widely when the Fortran 90 programs were compared to Fortran 77.

Machine	Language	Compiler	Number of Particles	Time (seconds)
	One-Dimensional Program			
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
	Two-Dimensional Program			
IBM RS6000	Fortran 90	IBM xlf90	327,680	526.71
IBM RS6000	Fortran 77	IBM xlf	327,680	549.23
IBM RS6000	C++	IBM xlC	327,680	667.00
	Three-Dimensional Program			
IBM RS6000	Fortran 77	IBM xlf	294,912	3230.06
IBM RS6000	C++	IBM xlC	294,912	3812.00

TABLE 7.4: Sequential Performance Characteristics for various programs on the IBM RS6000.

Additional Sequential Performance Comparisons The performance of the object-oriented one-dimensional and two-dimensional sequential Fortran 90 programs are compared to Fortran 77 and C++ versions in Table 7.4 for a larger problem. (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 contributed a performance overhead to the one-dimensional Fortran 90 program.

The two-dimensional Fortran 90 program (c.f. table 7.4) based on the model of figure 4.10 on page 92 slightly outperformed the Fortran 77 version in this larger problem while the C++ version was very competitive. The C++ three-dimensional program from the model of figure 3.5 on page 42 is also competitive with the Fortran 77 version.

Features	Intel Paragon	IBM SP2	Cray T3D
Processor Power	100 MFlops (single) 75 MFlops (double)	266 MFlops	150 MFlops (peak)
Network Speed	175 MB/sec	40/80 MB/sec (u.s.)	300 MB/sec

TABLE 7.5: Paragon XP/S, SP2 & T3D Basic System Characteristics (From Spec. Reports).

7.3 Parallel Simulation Results and Performance

The instability experiment measured the field, kinetic, and total energies of the system at each simulated time step. Since the original Fortran 77 codes have been well benchmarked [7], we will continue to restrict our performance overview to arbitrarily selected cases across the machines of interest—which are significantly smaller than production-oriented simulations. These results are only intended to illustrate how this code performs in Fortran 77, Fortran 90, 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 interconnection speed, in table 7.5. For the purposes of profiling we will continue to use small problem size examples.

In Table 7.6 we show processor simulation results for a few million particles across various simulation dimension sizes. Additional simulation comparisons are shown in Table 7.7 where these programs are based on the model in figure 3.7 on page 54. 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. Work performed in this area for sequential PIC codes have reported up to 90% of Fortran 77 efficiency [64].

The C++ version appears more competitive as more processors are used since the

Machine	PEs	Language & MP Library	Number of Particles	Time (sec)
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 7.6: Paragon XP/S, SP1/SP2 Multi-Million Particle Parallel Performance Characteristics.

problem size remains fixed as illustrated in Figure 7.1 on page 162. 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.

An object-oriented parallel Fortran 90 program for the SP2 (in two-dimensions) based on the model in figure 4.12 on page 95 has been written. The performance of this code, compared to the parallel Fortran 77 and C++ versions, is illustrated in table 7.8 on page 163 where the three-dimensional C++ program of figure 3.8 on page 56 is used.

Table 7.9 on page 164 compares the parallel performance of the two-dimensional Fortran 77, Fortran 90 and C++ programs on the IBM SP2 using 4 processors. The Fortran 77 and C++ programs use the IBM MPL message passing library, while the Fortran 90 program uses MPI. The C++ program is based on the two-dimensional particle vector model.

PEs	*Execution Time (seconds)*					
	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	1542.64	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	
PEs	*Execution Time (seconds)*					
	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 7.7: Paragon XP/S and SP1/SP2 Fixed Problem Size Parallel Performance Characteristics.

The `gprof` profiling tool returns data from each processor. The variance on a per processor basis was small, however, so table 7.9 only provides information from an arbitrarily selected processor. Additionally, while gprof can combine data from multiple nodes into a single profile, this information is accumulated, making comparison to the actual running time not very useful. The copy constructors for the `ChargedParticle` and `Vector2D` objects were called 6,668,096 and 106,492,000 times each for the 20,480 and 327,680 particle simulations respectively. We accumulated the performance statistics for these calls and added them to table 7.9 under `Copy Constructors`.

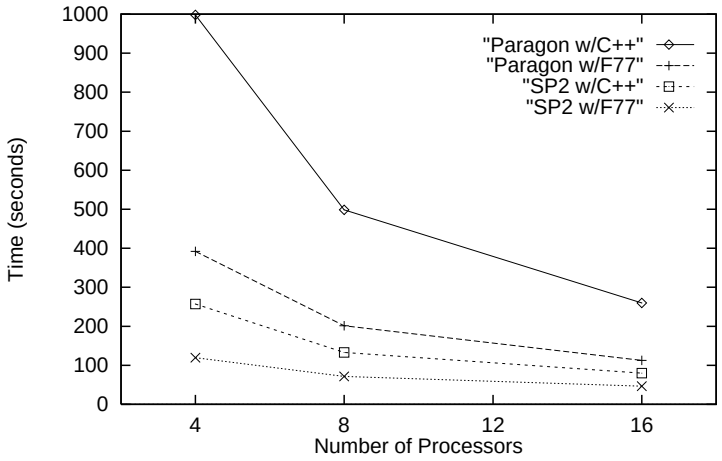


FIGURE 7.1: Paragon & SP2 Two-Dimensional Fortran 77 and C++ Execution Profile for a Fixed Problem Size.

Again, the Fortran 90 object-oriented program is very competitive with the Fortran 77 version, while the C++ program falls behind by about a factor of two. The major message passing routine (`Update Positions`) which moves particles across processors is fairly competitive among all the programs. However, the intensive `Advance` and `Deposit` routines in C++ do not compete with the Fortran programs. Even the `FFT` routine has fallen behind in C++, but this still represents a small portion of the performance problem.

In table 7.10 on page 165 we show performance results for the three-dimensional C++ program of figure 5.16 on page 123, where load balancing is not applied. The overhead of the mapping algorithm is fairly low when the performance is compared to omitting the algorithm. The mapping algorithm will be applied every time step in the worst possible manner since the static Fourier field partitions match the static electric and charge density field partitions. This implies that there are no local communication steps, since the mapping algorithm described in Chapter 5 will always communicate field data, even to itself. While it is possible to examine the communication operations allowing self-messages not to be transmitted, this introduces an inconsistency in the algorithm and it only occurs in this special instance. Therefore, this check was not introduced to preserve code clarity.

Machine	Language	Compiler	Number of Particles	Time (seconds)
	Two-Dimensional Program			
IBM SP2	Fortran 77	IBM xlf	3,571,712	195.08
IBM SP2	Fortran 90	IBM xlf90	3,571,712	202.88
IBM SP2	C++	IBM xlC	3,571,712	359.00
	Three-Dimensional Program			
IBM SP2	Fortran 77	IBM xlf	7,962,624	1564.27
IBM SP2	C++	IBM xlC	7,962,624	2797.00

TABLE 7.8: Performance characteristics for 2D and 3D two-stream beam-plasma instability experiment on IBM SP2 (AIX 4.1) with 3.5 million and 8 million particles on 32 processors.

The performance of the C++ programs is poor compared to the Fortran 77 versions. When comparing the performance of the code designed from figure 5.16 on page 123, much of the overhead is probably associated with the Fortran 90-style dynamic array operations introduced into the C++ programs. While the array classes have been designed with efficiency in mind, indexing overloaded multidimensional array operations on template class arguments often involves resolving a sequence of high-level overloaded operators to perform low-level operations.

For example, the following code segment (which uses message passing to move the complex Fourier field into the real electric force field) contains a number of overloaded statements. Each of these statements may cause a series of events to occur.

```
void Fields3D::DLBCmplxToElectricField( VectorField3D<float>& efield,
                                        const VPMachine& vm )
{
TransportBuffer3D< Fpt3D<float> > rgbuf( GEOM_X, GEOM_Y, GEOM_Z );
// code omitted above and below...
efield( x, y, OFFSET + z ) = rgbuf( x, y, z );  // Innocent statement
}
```

The operation indicated moves three-dimensional field components from the receive

	ALL OPTIMIZED (-O) Execution Time (seconds)					
	Fortran 77		**Fortran 90**		**C++**	
	total time without (with) profiling instrumentation					
	9.22	(9.53)	**9.86**	(10.26)	**20.00**	(39.62)
Operations	% total time in function and seconds in function alone					
Advance	28.60%	**2.75**	26.50%	**2.71**	15.20%	**6.02**
Deposit	16.30%	1.57	16.70%	1.70	8.40%	3.32
Update Positions	5.80%	0.56	8.10%	0.83	1.60%	0.62
FFT	5.10%	0.49	3.50%	0.36	4.80%	1.91
Poisson	0.10%	0.01	0.60%	0.06	0.20%	0.07
itrunc	17.00%	1.63	16.40%	1.67	4.00%	1.58
mcount	1.60%	0.15	1.00%	0.10	**41.90%**	**16.61**
Copy Constructors					**8.50%**	**3.36**
	20,480 Particles and 4 Processors					

	Fortran 77		**Fortran 90**		**C++**	
	total time without (with) profiling instrumentation					
	114.31	(117.14)	**117.49**	(121.51)	**249.00**	(516.99)
Operations	% total time in function and seconds in function alone					
Advance	36.10%	**42.33**	33.80%	**41.12**	19.30%	**99.65**
Deposit	20.30%	23.73	21.50%	26.13	10.80%	55.82
Update Positions	8.80%	10.32	8.70%	10.57	2.00%	10.59
FFT	1.60%	1.92	1.70%	2.08	1.60%	8.13
Poisson	0.10%	0.11	0.10%	0.13	0.10%	0.47
itrunc	26.30%	30.86	23.80%	28.86	4.80%	24.70
mcount	1.40%	1.67	1.60%	1.98	**44.10%**	**228.16**
Copy Constructors					**11.30%**	**58.52**
	327,680 Particles and 4 Processors					

TABLE 7.9: Parallel two-dimensional IBM SP2 Performance Comparison of Fortran 77 (IBM xlf), Fortran 90 (IBM xlf90) and C++ (IBM xlC).

PEs	Language	Compiler	Number of Particles	Time (seconds)
8	Fortran 77	IBM xlf	294,912	164.11
16	Fortran 77	IBM xlf	294,912	87.05
16	Fortran 77	Paragon f77	294,912	393.42
	Mapping Algorithm Applied...			
8	C++	IBM xlC	294,912	433.07
16	C++	IBM xlC	294,912	232.39
	Mapping Algorithm Not Applied...			
8	C++	IBM xlC	294,912	421.56
16	C++	IBM xlC	294,912	227.44
16	C++	Paragon CC	294,912	1799.00
	Vector Particle Program Model...			
8	C++	IBM xlC	294,912	381.00
16	C++	IBM xlC	294,912	185.00
16	C++	Paragon CC	294,912	1483.62

TABLE 7.10: Three-Dimensional parallel programs for IBM SP2 and Intel Paragon in the beam-plasma experiment. The modern class design, where the scanning partition mapping algorithm is used without load balancing, is compared to the same model without the mapping algorithm and the vector program model.

buffer into the electric field object within a loop, where each element has six words of data. The operation must perform the following functions:

1. Access `T& operator()` for the template object `rgbuf` which converts a three component address into a linear storage address. The `TransportBuffer3D<T>` template class handles the storage allocation and management.

2. Access `T& operator()` for the template object `efield` performing a similar conversion into dynamic storage used by the `VectorField3D<T>` template class.

3. The `VectorField3D<T>` template class dynamic storge is provided by a pointer object from the `DynamicMatrix3D<T>` template class. Thus, the `T& operator()`

Problem	PEs	Compiler	Number of Particles	Balancing Applied?	Time (seconds)
Gravitation	16	IBM xlC	294,912	No	777.12
Gravitation	16	IBM xlC	294,912	Yes	1330.42

TABLE 7.11: The effect of load balancing in the gravitational problem.

from the `DynamicMatrix3D<T>` class resolves the three-dimensional index address into its dynamic linear storage management system.

4. Since a field template object of type `Fpt3D<float>` is used as the lowest level template argument, the assignment operator (`Fpt3D<T>& operator=`) is called to perform memberwise assignment for the field components in `rgbuf` and `efield`.

5. Memberwise assignment of field components occurs, completing the operation above for a single iteration of the assignment loop.

While the dynamic storage features have obvious advantages, particularly when fields must be dynamically manipulated, the overhead associated with this simple assignment statement is obviously apparent. The performance of the vector particle program model based on the hierarchy of figure 3.8 is much more efficient because the field modeling is less sophisticated, it's static, and the details of the vector template class implementation are elementary in comparison.

The Load Balancing Programs Since the load balancing programs were written in C++, they can only be compared to themselves. The free expansion experiment only requires balancing in a minor way, so examining this problem is not very informative. The gravitational problem does contain regions of large density variations. For our purposes, we can compare the effects of load balancing to the performance with fixed partitions.

Table 7.11 shows the performance of the C++ load balancing code for a gravitational problem on the IBM SP2. Notice that the performance of the code which uses static partitions is *better* than the code which load balances the partitions. Actually, this is not all that surprising. Using 16 processors, the fields are partitioned into 16 components. In Chapter 5 we mentioned that our load balancing algorithm computes the particle density function, used in repartitioning, by mapping every particle to a global density distribution array. This is expensive. If we modified the scheme to use the existing charge distribution then examining every particle would not be necessary, only the field would be inspected, improving the performance [12]. Additionally, we decided to check if repartitioning is needed after the particle positions were updated since the exact number of particles per processor is only known at this time. This implies that when load balancing is required, particles must again be moved among processors before the PIC loop can continue. This essentially doubles the amount of communication required when load balancing is applied, adding to poor performance. Nevertheless, the major contributor to poor performance is the dynamic resizing, creation, and destruction of fields that occurs since partitions are modified frequently as the particle distribution changes. The mapping algorithm between the static and moving fields is fast, but the quality of partitions is poor since simple partitioning schemes were applied.

Similarly, many of the objects used including the particle species, various fields, and transport buffers for object based message passing, automatically resize themselves when memory requirements cannot be satisfied or when the geometry of the object has changed. These operations are also implemented at the lowest level definition for components of these complex objects where the calling sequences are comparable to the object assignment example recently discussed. This also contributes to the performance overhead.

Obviously, load balancing should be applied only when the overhead of managing the run-time system does not outweigh using a static partitioning scheme. Most

charge-neutral plasma experiments tend to have self-balancing characteristics, so load balancing may not always be necessary. In our examples, balancing was beneficial in maintaining nearly equal portions of work among processors, but the overhead involved in finding the density distribution exceeded the benefit of repartitioning. This is encapsulated into the field management code, so more efficient schemes which do not inspect particle positions explicitly can be introduced to help improve performance. Advanced memory management schemes to reuse allocated storage space would also help. As mentioned in Chapter 5, our interest involved exploring a context for object based load balancing, not on improving existing balancing methods, so a simple balancing scheme was introduced for this purpose. The monitoring technique did not contribute to the performance effects.

7.4 Measuring the Performance Effects of Object-Oriented Abstractions in Fortran 90

In the spirit of the *Stepanov C++ Benchmark* [59], we have been investigating the performance effects of using object-oriented features in Fortran 90 programs. The Stepanov benchmark computes an *abstraction penalty* that represents the cost of applying increasing levels of abstraction in a Fortran-style loop computation. A series of tests, written using STL-based iterator constructs in C++, are used. Ideally, compilers should remove the overhead associated with using abstraction features so that the subsequent tests perform as well as the initial test.

The Stepanov benchmark brings attention to an important issue, *abstraction modeling may degrade performance.* Our benchmark for Fortran 90 addresses the same issue. A series of tests are performed, however, our tests more appropriately reflect how actual programs may be developed in Fortran 90. We return three kinds of abstraction measurements that indicate the effects of the tests: abstraction of Fortran 90 array-types, inheritance abstraction, and the overall abstraction.

Illustration of base test multiplication routine

```
SUBROUTINE base_multiply(ix,iy,iz,iter,isize)
   DIMENSION ix(isize), iy(isize), iz(isize)
   INTEGER, INTENT(in) :: iter, isize
   INTEGER :: i, iterations              ! Variable declarations
   DO i=1,isize                          ! Initialize...
      ix(i) = 10 ; iy(i) = 2
   END DO
   DO iterations=1,iter                  ! Multiply Op...
      DO i=1,isize
         iz(i) = ix(i) * iy(i)
      END DO
   END DO
END SUBROUTINE base_multiply
```

FIGURE 7.2: Structure of the Fortran 77-based multiplication test that is replicated with various Fortran 90 array-syntax and object-oriented abstractions.

The Benchmark Tests This benchmark currently performs twenty tests measuring the effects of using every kind of Fortran 90 style array—along with the use of advanced features including modules, use-association, pointers, and object-oriented features including encapsulation, and inheritance by composition and sub-typing. Every tests is based upon a Fortran 77 style component-wise array multiplication, as seen in figure 7.2.

As in the Stepanov test, which performs a different computation, our benchmark reports on a kernel-by-kernel basis the run-time ratio of the current abstraction test to the `base_multiply` test. All of the tests that are performed involve the component-wise multiplication in some form. For example `test 11`, which multiplies two static arrays defined by a Fortran 90 derived type, allows arrays to be created and initialized from a module:

```
! Create array objects from STATIC_ARRAY_MODULE and initialize
       TYPE (static_array) :: sx, sy, sz
       call init(sx, iter) ; call init(sy, iter) ; call init(sz, iter)
```

Then, the actual multiplication test is performed:

```
sz = sx * sy
```

where the '*' operator has actually been overloaded to perform the multiplication on the derived types. Note that the initialization of these arrays also includes the number of iterations of `test 1` style multiplications. Part of the module definition is shown in figure 7.3 on page 171.

The Benchmark Results Figure 7.4 on page 172 shows a sample run of the Fortran 90 benchmark program. The results are generally consistent regardless of the number of iterations applied. The benchmark has been applied to a variety of compilers, some of which are in the beta-test stage. Due to release restrictions the results of these tests cannot be described. We can mention, however, that some compilers fail to build the benchmark while the performance of other depends greatly on the optimization switches applied.

The overhead of applying array syntax abstractions is very low, while the usage of inheritance-based features is also small. This implies that both of these techniques can be used to improve the modeling of programs without major concern for performance degradation effects. The abstraction measurement ratios computed do depend on how well the compiler performs on `test 1`. This benchmark program will continue to grow with the addition of new tests including, array subsection operations and other object-oriented features including dynamic dispatching. Detailed descriptions of the specific tests performed are available [42].

7.5 Commentary

Now and then statements are made claiming that object-oriented codes can run faster than procedural codes, where "object-oriented" is synonymous with C++ and "procedural" is synonymous with Fortran. In general, we have found the performance of Fortran 77 to remain ahead of C++ and C, while the performance of object-oriented

Illustration of module and multiplication routine from test 11

```
MODULE static_array_module
   USE global_data
   IMPLICIT NONE
   SAVE
   TYPE static_array        ! Array definition and loop iterations
       INTEGER :: iter
       INTEGER, DIMENSION(gdim) :: p
   END TYPE static_array
   INTERFACE init           ! Define constructor interface
       MODULE PROCEDURE sa_create
   END INTERFACE
   INTERFACE operator(*)    ! Overload multiplication operator
       MODULE PROCEDURE sa_mult
   END INTERFACE
CONTAINS
   SUBROUTINE sa_create(this,range)        ! Constructor
      TYPE (static_array), INTENT(out) :: this
      INTEGER, INTENT(in) :: range
      this%iter = range                    ! Assign components
      this%p = 10
   END SUBROUTINE sa_create
   TYPE (static_array) FUNCTION sa_mult(x,y)
      TYPE (static_array), INTENT(in) :: x, y
      INTEGER :: iterations
      DO iterations=1,x%iter               ! Multiply operation...
            sa_mult%p = x%p * y%p
      END DO
   END FUNCTION sa_mult
END MODULE static_array_module
```

FIGURE 7.3: Structure of the of static array module multiplication test. Array objects can be created from the module where overloaded operations—in combination with array syntax—are applied in multiplication.

Sample Benchmark Run on IBM RS/6000

```
Iterations (2000 minimum) : 2000
Working ..................
```

Test	Time	Base Ratio	Test	Time	Base Ratio
1	0.290	1.000	11	0.380	1.310
2	0.280	0.966	12	0.270	0.931
3	0.300	1.034	13	0.620	2.138
4	0.320	1.103	14	0.640	2.207
5	0.300	1.034	15	0.290	1.000
6	0.400	1.379	16	0.620	2.138
7	0.280	0.966	17	0.570	1.966
8	0.270	0.931	18	0.290	1.000
9	0.280	0.966	19	0.580	2.000
10	0.430	1.483	20	0.590	2.034

```
Array Syntax Abstraction Measure:    1.178
Inheritance Abstraction Measure:     1.500
Overall Abstraction Measure:         1.303
```

FIGURE 7.4: Abstraction benchmark results applied to IBM RS/6000 with the IBM xlf90 Fortran 90 compiler (optimization -O2 with AIX 4.1). Note that `test 6` (automatic array test with use-association) always returns 0.000 as the execution time, regardless of the iteration length. The time indicated for that test is a loose approximation based on `test 5`. Abstraction measurements close to 1.000 are desired.

Fortran 90 programs have been better than C++ and very competitive with Fortran 77, particularly for parallel programming. The abstraction capabilities of C++, for those most experienced with this language, may offset the performance differences compared to Fortran. This is especially true if features such as templates are used which currently are not part of Fortran 90. Additionally, it could be misleading to generalize about performance figures for all kinds of codes based on our experience since these programs contain features that may favor the years of optimization effort already present in Fortran compilers. Nevertheless, all indications seem to imply that for numerical loop intensive codes with large memory and computation requirements,

Fortran remains more efficient than C++.

Many researchers are successfully working on improving the performance of C++ compilers by examining how language constructs are used [52], including the Stepanov [59] and OOPACK [53] benchmarks. Unfortunately, most of these tests examine programming features in isolation without the context of a larger problem. This work is extremely beneficial and important, however, leading to improvements in C++ compiler technology. Indeed, Robison suggests that the notion of an *abstraction penalty*, a terminology introduced by Stepanov used to gauge the efficiency of programs that use abstraction modeling features, will " ... affect how programmers use them or even *whether* programmers will use the language at all [54]." He brings attention to the performance penalties lack of optimization for small objects, virtual functions, and exceptions may produce.

Additional, very interesting work, involves the use of C++ templates to represent parse trees of mathematical expressions as types—to reduce the creation of temporaries and additional loops in chained expressions. The first work described on this topic is by Vandevoorde [65] in the `valarray<Troy>` numerical array library implementation. This approach allows expression evaluation to be delayed through the creation of static expression trees. A similar technique, published shortly afterward by Veldhuizen [67] introduces "Expression Templates" which pass expression-types as template parameters and function arguments. Delayed evaluation with parse trees are also used, leading to performance benefits. Member templates are required to exploit the fullest potential of this work—this feature has only recently become available on commercial compilers [52]. Both implementations introduce a workaround for the lack of member templates that induce a small fixed cost in performance, and less general applicability [66]. This kind of research should help reduce the abstraction penalty of applying sophisticated mathematical constructs for C++ compilers capable of using the technique effectively.

We only introduced virtual functions in the load balancing codes and their use

did not introduce serious performance problems compared to issues already addressed. The other benchmarks do not use virtual functions at all, yet poor compiler performance was still reported. Our programs contain a mixture of very large numbers of small objects—in the millions, a small number of extremely large objects—the fields for example, and object collections which vary in size and extent—the fields and species. To alleviate any potential penalty from accessing small objects through member functions which might not be in-lined, data visible for read/write access was always public. This does not lower the level of abstraction and it produced performance improvements in the Fortran 90 programs where in-lining was not always available.

Given the variety, extent, and level of interaction of abstract components in these programs, determining the best course of action to improve performance may be difficult. A methodology for determining the absolute performance of any object-oriented program is still open. This kind of study would be very useful since there may be inherent limits to performance when abstraction is introduced. This would help to establish if expecting the performance of a C++-based program to match or exceed a Fortran-based program is an appropriate question to consider.

Chapter 8

Discussion, Conclusions, and New Directions

The importance of paradigm-related studies is reviewed as researchers contemplate what programming languages and approaches to consider for application development. Additionally, we discuss how this research may influence other popular contemporary topics in object-oriented scientific programming.

8.1 The Impact of Paradigm Studies on Modern Software Development

Performance and programmability are the two most important factors mentioned in software support for scientific programming. Comparative analysis allows one to examine the state of software technology, provided an application typical of how various languages are used is selected. The study of modern language paradigms renews our insight into how applications are developed, while critiquing current programming techniques against new directions proposed to enhance advanced software design.

Object-oriented programming simultaneously makes application development easier and more complex since development and usage of objects occurs over a variety

of levels. Given a problem, an abstract model must be created which is then mapped onto the features of a language that supports object-oriented programming. Application programming begins with a high-level view of major operations on simulation objects. Infrastructure development forms the next deeper level, supporting the high-level view. This involves the design of classes with support for their interaction. Lower levels feature the implementation of class components that have an object-oriented view of their own—which may differ from the high-level view they support. Understanding how languages support development of abstractions across these levels is important in scientific programming since computational abstractions often exist simultaneously across levels. Paradigm studies among languages bring greater understanding to this issue, including the effects this may have on performance and abstraction modeling, based on constructs provided by different languages.

The objective of this thesis has been to apply a systematic approach to the study of language paradigms in scientific programming on high performance computers. Current and future applications will require greater flexibility in design and organization while providing performance good enough to support Grand-challenge software programming. The effects of our study are already prevalent throughout national laboratories, universities, and industrial corporations by a renewed interest in paradigm comparison issues. We review the major goals and impact of this research in this concluding chapter.

8.2 Review of Thesis Research

The major original contribution of this work is that comparative paradigm-related issues have been addressed in object-oriented programming for a single scientific application using existing and emerging languages on scalar and parallel architectures. One of the main objectives was to bring new ideas and concepts to research, academic, and industrial laboratories with an evaluation of their effects when compared

to traditional ways of implementing software. The most important contribution is that our objective study provides experiences that often cannot be realized in organizations due to various constraints that may be imposed, whether they are technical or managerial. A formal description of the specific contributions has been presented in Chapter 1, however, we comment on many of these issues in the following sections.

Object-Oriented Development of Plasma Programs Various designs for object-oriented scalar and parallel plasma particle-in-cell codes were designed and presented. These were implemented from the initial Fortran 77 programs using object-oriented techniques in C++ and Fortran 90. A discussion of how a scientific computation is analyzed and modeled in an object-oriented fashion, with progressive development and refinement of the models including the consequences of implementation decisions, was presented. This analysis provides insight into the design process and the factors that influence design decisions.

The C++ language provided many powerful mechanisms for development of advanced plasma programs with abstraction modeling. The template features were very useful while popular object-oriented features, such as inheritance, saw limited usefulness. Object models which support interactions among abstractions with multiple views were very easy to model. While the goals and design organization of the C++ programming language are sound [61], specific compiler implementations make many of these benefits difficult to use effectively. Our development on the IBM SP and Intel Paragon systems and compilers have been largely successful, but the Cray T3D compiler problems did not allow for useful development work on this machine using the C++ programming language.

The need to modernize languages such as Fortran 77 has led to the development of Fortran 90 and High Performance Fortran (HPF), which leverage the infrastructure of existing programs and knowledge of experienced users with new techniques to simplify code development. Although new features such as array operations and data

distribution directives are very beneficial, supporting object-oriented concepts such as abstraction, encapsulation of ideas, and enhanced collaboration appeared to remain inaccessible to the Fortran community.

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

Additional material on object-oriented programming in Fortran 90, beyond the scope of Chapter 4, will be available in future publications [44]. Some of our modern work with this language falls beyond the scope of this thesis—length concerns restrict our ability to present every detail. This includes features like code migration. The migration of Fortran 77 codes to Fortran 90, using language free abstractions, was straightforward. Code migration is important since it provides a way to modernize existing parallel applications within a familiar context. Similarly, HPF is designed around Fortran 90 so it will be possible to combine the object paradigm with this language as HPF compilers mature.

Object Paradigms for Dynamic Load Balancing Load balancing is an important requirement in scientific parallel programming. We have designed an object-

oriented instrumentation technique which allows specific objects to be monitored on a continuous basis, in an implicit fashion, based on normal object usage. This simplifies the context of how load balancing can be handled for object-oriented scientific programs.

Our experience with the plasma programs found that the technique was effective, providing useful information to the load balancer in a non-obtrusive manner. However, the load balancing algorithm applied was simplistic; the overhead in computing new partition addresses was more expensive than necessary, although dynamic memory management of large fields was the major cause of poor performance. The technique was applied to a collisionless free expansion and gravitational experiment which exhibit two different forms of irregularity.

Performance Comparisons Among Language Paradigms A detailed analysis of the performance of various object-oriented models for programs written in Fortran 90, C, and C++ were compared across compilers and architectures on scalar and parallel machines. The Fortran 77 performance usually outperformed Fortran 90, but the results were always comparable. The C++ performance typically was twice as slow as Fortran 77 for all simulation problems and machine environments.

Many comparative issues were addressed for the first time, including performance of Fortran 77, Fortran 90 and C++ for a single complete scientific application on workstations and supercomputers. Careful implementation of library functions was identified as an issue of concern, while features thought to provide performance improvements, such as in-lining, sometimes had a minimal effect on performance. The introduction and overhead of using complex operations on abstractions can lead to performance effects as operators are pushed through an abstraction hierarchy to the base level implementation. Techniques to optimize code associated with these operations could lead to significant performance enhancements.

8.3 Final Commentary and New Directions

It is now safe to move toward modern programming paradigms, in particular object-oriented methods in scientific programming. This safety comes in a variety of forms, mainly that developers have a larger selection among languages. Fortran 90 can be used very effectively to program in this paradigm. For those who have other requirements beyond highest performance, such as graphics, interfaces to devices, and so on, C++ can be used with (a perhaps manageable) performance penalty. In each case, users should be aware of the state of compiler technology and make their decisions not based on hearsay, but on empirical evidence and comparative analysis. This is the kind of information this thesis provides.

There are many new directions to explore from topics covered within this thesis research. This includes the *integration of related codes* that support the design of object models, and frameworks for simulations which share common features. The *design effects of abstraction* is an interesting and serious issue that can benefit from further study. Although the object-oriented methodology is specified independently of a particular programming language, we have learned that features of Fortran 90 and C++ can affect the modeling of the problem. In particular, when comparing or converting from one language to the other, it may not be possible to maintain identical object-oriented representations. Object-Oriented *abstractions also affect the efficiency* of scientific computations. Since the manner in which the program is modeled can affect performance, studying strategies that maximize performance based on various representations is required. We have already seen examples of this issue, but new efforts geared toward minimizing the performance impact of abstraction are needed. As our research has illustrated, such approaches should be experimental.

The notion of *design patterns*, which try to capture the experience of recurring successful designs, have become popular. Many scientific simulations share common features that most likely contain patterns. Discovery of these features, which could then be modeled in an abstract way, may ease the development of templates and

extension of scientific software components to related problems. Although we do not agree with all of the claims offered by the patterns community, this paradigm must be investigated further to access its applicability in scientific programming. One obvious pattern appropriate to parallel scientific programming is the guard region. This concept is used often to minimize message passing of partitioned data. Many other important patterns certainly exist.

As mentioned in the preface, the impact of this study has found it way into many organizations. Researchers are curious about comparisons between Fortran 90 and C++ while corporations and laboratories are re-thinking their software design strategies based on early access to this research. Additionally, much of the design work in object-oriented modeling for the plasma simulation problems is leading into new efforts in Fortran 90 development for the next generation of modern plasma simulation programs.

Readers may now confidently consider what aspects of various paradigms are useful or inappropriate; how should a scientific computation be designed to take advantage of object-oriented techniques; what are the benefits and pitfalls of the methodology; what languages are most appropriate to allow researchers who are not familiar with object-oriented programming, but have significant backgrounds in other languages, to enter the field? The significance of this work is broad since it addresses concerns programmers must address when evaluating the usage of new paradigms for current and future development projects. Since paradigms affect scientific programming directly, the impact of this study has been immediate. These issues are relevant as developers in academia, government, and industry make decisions regarding programming approaches for current and future scientific development projects. There is a need for a comprehensive and serious study of the options available when considering languages for important projects, particularly when issues such as legacy codes and the need to design for extension are considered. We hope that our analysis of programming paradigms will positively affect application development.

Literature Cited

[1] C. K. Birdsall and A. B. Langdon. *Plasma Physics via Computer Simulation.* The Adam Hilger Series on Plasma Physics. Adam Hilger, New York, 1991.

[2] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Yang. *Distributed pC++: Basic Ideas for an Object Parallel Language.* Irisa, U. of Rennes and Department of Computer Science, Indiana University, January 1993.

[3] G. Booch. Object-Oriented Development. *IEEE Transactions on Software Engineering*, SE–12(2):211–221, February 1986.

[4] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers: Design, Implementation, and Performance Results. In *Proc. Supercomputing '93*, pages 351–360, Portland, Oregon, November 15–19 1993. IEEE Computer Society.

[5] B. I. Cohen, D. C. Barnes, J. M. Dawson, G. W. Hammett, W. W. Lee, G. D. Kerbel, J.-N. Leboeuf, P. C. Liewer, T. Tajima, and R. E. Waltz. The numerical tokamak project: simulation of turbulent transport. *Computer Physics Communications*, 87(1&2):1–15, May II 1995.

[6] W. Cook, W. Hill, and P. Canning. Inheritance Is Not Subtyping. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, pages 125–135, January 1990.

[7] V. K. Decyk. Skeleton PIC Codes for Parallel Computers. *Computer Physics Communications*, 87(1&2):87–94, May II 1995.

[8] V. K. Decyk, C. D. Norton, and B. K. Szymanski. Experiences with Object Oriented Parallel Plasma PIC Simulation. In *Proc. Computing in High Energy Physics (CHEP '95)*, Singapore, Sept. 18–25 1995. Lafex/Fermilab, World Scientific. Invited Plenary Talk at Computing in High Energy Physics '95 Conference, Rio de Janeiro, Brazil.

[9] V. K. Decyk, C. D. Norton, and B. K. Szymanski. Introduction to Object-Oriented Concepts Using Fortran 90. Technical Report PPG–1560, Institute

of Plasma and Fusion Research, UCLA Dept. of Physics and Astronomy, Los Angeles, CA 90095-1547, July 1996. Submitted to Computers in Physics.

[10] J. Denavit. Collisionless plasma expansion into a vacuum. *Phys. Fluids*, 22(7):1384–1392, July 1979.

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

[12] R. D. Ferraro, P. C. Liewer, and V. K. Decyk. Dynamic Load Balancing for a 2D Concurrent Plasma PIC Code. *J. Computational Physics*, 109(2):329–340, December 1993.

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

[14] K. Fisher and J. C. Mitchell. What is an Object-Oriented Programming Language. From "Notes of typed object-oriented programming," Proc. Theoretical Aspects of Computer Software, Springer LNCS 789, 1994, pages 844-885, June 1995.

[15] D. W. Forslund, J. S. Junkins, and C. A. Wingate. WAVE++: A Distributed, Object-Oriented Plasma Simulation Code. In *Proc. US-Japan Workshop on Advances in Simulation Techniques Applied to Plasmas and Fusion*, pages 1–4, Los Angeles, California, September 26–28 1990.

[16] I. T. Foster and K. M. Chandy. *Fortran M: A Language for Modular Parallel Programming*. Argonne National Laboratory and California Institute of Technology, June 1992.

[17] I. T. Foster and D. W. Walker. Paradigms and Strategies for Scientific Computing on Distributed Memory Concurrent Computers. In *Proc. Conference on High Performance Computing*, pages 252–257, La Jolla, California, April 11–15 1994. IEEE Computer Society.

[18] General Electric Company, Advanced Concepts Center, King of Prussia, PA. *OMTool User Guide*, release 1.2 edition, February 1993.

[19] A. S. Grimshaw. Easy-to-Use Object-Oriented Parallel Processing with Mentat. *IEEE Computer*, 26(5):39–51, May 1993.

[20] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press ISBN 0-262-57104-8, Cambridge, Massachusetts, 1994.

[21] S. W. Haney. Is C++ Fast Enough for Scientific Computing? *J. Computers in Physics*, 8(6):690–694, Nov/Dec 1994.

[22] S. W. Haney and J. A. Crotinger. C++ Proves Useful in Writing a Tokamak Systems Code. *J. Computers in Physics*, 6(5):450–455, Sep/Oct 1991.

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

[24] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler Support for Machine-Independent Parallel Programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers and Run–Time Environments for Distributed Memory Machines*, pages 139–176. North–Holland, Amsterdam, 1992. Elsevier Science Publishers B.V.

[25] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles.* Institute of Physics Publishing, Philadelphia, 1994.

[26] J. K. Hollingsworth and B. P. Miller. An Adaptive Cost Model for Parallel Program Instrumentation. In *Proc. EuroPar'96*, Lyon, France, Aug 1996.

[27] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic Program Instrumentation for Scalable Performance Tools. In *Proc. Scalable High Performance Computing Conference*, pages 841–850, Knoxville, Tennessee, May 1994. IEEE Computer Society.

[28] D. Kafura and L. Huang. mpi++: A C++ Language Binding for MPI. In *Proc. MPI Developers Conference, Notre Dame, IN*, June 1995.

[29] L. V. Kale and S. Krishnan. Charm++: A Portable Concurrent Object Oriented System Based on C++. Technical Report UIUCDCS-R-93-1793, Department of Computer Science, University of Illinois, Urbana-Champaign, March 1993.

[30] V. Karamcheti and A. Chien. Concert - Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware. In *Proc. Supercomputing '93*, pages 598–607, Portland, Oregon, November 15–19 1993. IEEE Computer Society.

[31] S. R. Kohn and S. B. Baden. The Parallelization of an Adaptive Multigrid Eigenvalue Solver with LPARX. In *Proc. Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 552–557, San Francisco, California, February 15–17, 1995.

[32] G. A. Kohring. Dynamic Load Balancing for parallelized particle simulations on MIMD computers. *Parallel Computing*, 21:683–693, 1995.

[33] B. B. Kristensen and K. Østerbye. A Conceptual Perspective on the Comparison of Object-Oriented Programming Languages. *ACM SIGPLAN Notices*, 31(2):42–54, Feb 1996.

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

[35] P. C. Liewer, E. W. Leaver, V. K. Decyk, and J. M. Dawson. Dynamic Load Balancing in a Concurrent Plasma PIC Code on the JPL/Caltech Mark III Hypercube. In *Proc. Fifth Distributed Memory Computing Conference*, pages 939–942, 1990.

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

[37] L. Martin. Writing Optimizable Code using Fortran 90. Invited presentation at SHARE 85, Orlando, Fl, August 1995.

[38] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, Reading, MA, 1996.

[39] C. Neusius. A Concurrent Object-Oriented Programming Language and its Distributed Implementation. In *Proc. Fifth Distributed Memory Computing Conference*, pages 1074–1079, 1990.

[40] M. Nibhanupudi, C. Norton, and B. Szymanski. Plasma Simulation on Networks of Workstations using the Bulk Synchronous Parallel Model. In *Proc. Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, pages 13–22, Athens, Georgia, November 3–4 1995.

[41] O. Nierstrasz. A Survey of Object-Oriented Concepts. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 3–21. ACM Press and Addison–Wesley, 1989.

[42] C. D. Norton. Abstraction benchmark measurements for object-oriented Fortran 90 programming—a white paper. Unpublished Manuscript, September 1996.

[43] C. D. Norton. An Efficient Mapping Algorithm for Irregularly Coupled Fields in Load Balancing for Plasma PIC Simulations. Unpublished Manuscript, July 1996.

[44] C. D. Norton, V. K. Decyk, and B. K. Szymanski. High Performance Object-Oriented Programming in Fortran 90. Internet Web Pages, October 1996. http://www.cs.rpi.edu/~nortonc/oof90.html.

[45] C. D. Norton, V. K. Decyk, and B. K. Szymanski. On Parallel Object Oriented Programming in Fortran 90. *ACM SIGAPP Applied Computing Review*, 4(1):27–31, Spring 1996.

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

[47] C. D. Norton, B. K. Szymanski, and V. K. Decyk. Parallel Object Oriented Implementation of a 2D Bounded Electrostatic Plasma PIC Simulation. In *Proc. Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 207–212, San Francisco, California, February 15–17, 1995.

[48] Overview of JPL's Participation in NASA's HPCC Earth and Space Science (ESS) Project. Internal Document, 1994.

[49] S. Parkes, J. A. Chandy, and P. Banerjee. A Library-based Approach to Portable, Parallel, Object-Oriented Programming: Interface, Implementation and Application. In *Proc. Supercomputing '94*, pages 69–78, Washington, D.C., November 14–18 1994. IEEE Computer Society.

[50] J. V. W. Reynders. Object-Oriented Particle Simulation on Parallel Computers. In *15th International Conference on the Numerical Simulation of Plasmas*, pages 1B2 1–4, King of Prussia, Pennsylvania, September 7-9 1994.

[51] J. V. W. Reynders, D. W. Forslund, P. J. Hinker, M. Tholburn, D. G. Kilman, and W. F. Humphrey. Object-Oriented Particle Simulation on Parallel Computers. In *Object Oriented Numerics Conference*, pages 266–279, 1994.

[52] A. D. Robison. C++ Gets Faster for Scientific Computing. *J. Computers in Physics*, 10(5):458–462, Sep/Oct 1996.

[53] A. D. Robison. OOPACK: a Benchmark for Comparing OOP vs C-style programming. http://www.kai.com/oopack/oopack.html, 1996.

[54] A. D. Robison. The Abstraction Penalty for Small Objects in C++. Kuck and Associates, Champaign IL, 1996.

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

[56] E. Seidewitz. Object-Oriented Programming in Smalltalk and ADA. *ACM SIGPLAN Notices*, 22(12):202–213, December 1987. In Proc. OOPSLA 87 edited by N. Meyrowitz.

[57] P. H. Smith and S. M. Griffin, editors. *Workshop and Conference on Grand Challenges Applications and Software Technology*, Pittsburgh, Pennsylvania, May 4–7, 1993. GCW-0593.

[58] J. M. Squyres, B. C. McCandless, and A. Lumsdaine. Object Oriented MPI: A Class Library for the Message Passing Interface. POOMA '96 Conference, Sante Fe, New Mexico, 1996.

[59] A. Stepanov. KAI's Version of Stepanov Benchmark – Version 1.2. http://www.kai.com/C_plus_plus/benchmarks/ftp.html, 1996.

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

[61] B. Stroustrup. Why C++ is not just an Object-Oriented Programming Language. *OOPS Messenger*, 6(4):1–13, October 1995. Addendum to Proc. OOPSLA '95.

[62] B. K. Szymanski and C. D. Norton. Object Oriented Programming in Parallel Scientific Computing. *ACM SIGAPP Applied Computing Review*, Oct 1996. (in press).

[63] The HPC++ Working Group. HPC++ Whitepapers. Technical Report TR 95633, Center for Research on Parallel Computation, 1995.

[64] M. Turner. Experience with PIC-MCC and C++. In *15th International Conference on the Numerical Simulation of Plasmas*, King of Prussia, Pennsylvania, September 7-9 1994.

[65] D. Vandevoorde. valarray<Troy> an implementation of a numerical array. Internet Web Pages, February 1995. ftp://ftp.cs.rpi.edu/pub/vandevod/Valarray/ and ANSI X3J16 papers.

[66] D. Vandevoorde. Personal Communication. Electronic–Mail, October 1996.

[67] T. Veldhuizen. Expression Templates. *C++ Report*, 7(5):26–31, June 1995.

[68] J. P. Verboncoeur, A. B. Langdon, and N. T. Gladd. An Object-Oriented Electromagnetic PIC Code. Technical Report UCB/ERL M94/71, Electronic Research Laboratory, College of Engineering, UC Berkeley, Berkeley, CA 94720, September 1994.

[69] I. Wells. Personal Communication. Electronic–Mail, September 1996.

[70] R. D. Williams. DIME++: A Parallel Language for Indirect Addressing. Technical Report CCSF-29, CCSF, California Institute of Technology, Pasadena, CA, January 1993.

[71] G. V. Wilson and P. Lu, editors. *Parallel Programming Using C++*, Cambridge, Massachusetts, 1996. MIT Press. Scientific and Engineering Computation Series.

Appendix A

Plasma PIC Programming Source Segments

This section shows program abstracts for various segments of the plasma PIC applications beyond what is appropriate to include in the main text. Various portions of the codes written in Fortran 77, Fortran 90, and C++ are illustrated to enhance comparison among these languages. These programs generate diagnostics, and sometimes visualization data, in an output file—this data will be not presented. In general, only the main program interfaces are included.

A.1 Fortran 77 Scalar 1D Initialization Section

The following is the initialization section for the scalar Fortran 77 program, where the declaration of variables has been omitted for brevity.

```
c prepare fft tables
      isign = 0
      call fft1rx(q,t,isign,mixup,sct,indx,nx,nxh)
c calculate form factors
      call pois1 (q,fx,isign,ffc,ax,affp,we,nx)
c initialize density profile and velocity distribution
c background electrons
      do 110 j = 1, nx
      q(j) = 0.
  110 continue
      if (npx.gt.0) call distr1 (part,vtx,zero,npx,idimp,nx)
c beam electrons
      if (npxb.gt.0) call distr1 (part(1,npx1),vtdx,vdx,npxb,idimp,nx)
c deposit charge for initial distribution
      call dpost1 (part,q,qme,np,idimp,nx)
c add background ion density
      qi0 = -qme/affp
      do 170 j = 1, nx
      q(j) = q(j) + qi0
  170 continue
```

A.2 Fortran 77 Scalar 1D Loop Section

The following is the loop section for the scalar Fortran 77 program, where the declaration of variables has been omitted for brevity.

```
  500 if (nloop.le.itime) go to 2000
c transform charge to fourier space
      isign = -1
      call fft1rx(q,t,isign,mixup,sct,indx,nx,nxh)
c calculate force/charge in fourier space
      call pois1 (q,fx,isign,ffc,ax,affp,we,nx)
c transform force/charge to real space
      isign = 1
      call fft1rx(fx,t,isign,mixup,sct,indx,nx,nxh)
c particle push and charge density update
c initialize charge density to background
      do 1180 j = 1, nx
      q(j) = qi0
 1180 continue
      wke = 0.
c push particles
      call push1 (part,fx,qtme,dt,wke,idimp,np,nx)
c deposit charge
      call dpost1 (part,q,qme,np,idimp,nx)
c energy diagnostic
      wt = we + wke
      write (6,993) we, wke, wt
      itime = itime + 1
      go to 500
 2000 continue
```

A.3 C++ Scalar 1D Initialization Section

The following is the initialization section for the scalar C++ program.

```
// Create simulation objects
Plasma species;
Electron elec[NP];
ParticleDist backelec( 1.0F, 0.0F, SYSLEN_X, N_BKELE_X );
ParticleDist beamelec( 0.5F, 5.0F, SYSLEN_X, N_BMELE_X );
Grid grid;
EnergyDiag energy;
Timer time;

// Prepare FFT tables and initialize density profile
     grid.Setup( energy );
// Particle spatial/velocity distribution
     backelec.Maxwellian( elec );
     beamelec.Maxwellian( &elec[N_BKELE_X] );
// Deposit charge
     grid.DepositCharge( elec, Electron::charge, NP );
// Add background Ion density
     grid.AddIonDensity();
```

A.4 C++ Scalar 1D Loop Section

The following is the loop section for the scalar C++ program. The declaration of objects is shown in the initialization section.

```
for (int i = 0; i < N_STEPS; i++) {

// Calculate Force/Charge using FFT
     grid.CalcEField( energy );
// Charge density field update
     grid.InitChargeDensity();
     energy.ke(0.0);
// Push particles
     species.Advance( elec, grid, energy, NP );
// Deposit charge
     grid.DepositCharge( elec, Electron::charge, NP );
// Energy diagnostic
     energy.tote(energy.pe() + energy.ke());
     curOFile << "Time : " << i << endl;
     curOFile << energy;
}
```

A.5 C++ Scalar 1D Main Program (Initial)

```
main()
{
Plasma plasma;                                  // Create Objects
BackgroundElectron bkelec[N_BKELE_X];
BeamElectron bmelec[N_BMELE_X];
Grid grid;
EnergyDiag energy;

   grid.Setup( energy );     // Initialize and construct grid

   plasma.DenVelDist( bkelec, BackgroundElectron::thermal_vel_x,
               BackgroundElectron::drift_vel_x, N_BKELE_X );
   plasma.DenVelDist( bmelec, BeamElectron::thermal_vel_x,
               BeamElectron::drift_vel_x, N_BMELE_X );

   grid.DepositCharge( bkelec, Electron::charge, N_BKELE_X );
   grid.DepositCharge( bmelec, Electron::charge, N_BMELE_X );
   grid.AddIonDensity();

   for (int i = 0; i < N_STEPS; i++) {
        grid.CalcEField( energy );
        grid.InitChargeDensity();
        energy.ke(0.0);
        plasma.Advance( bkelec, grid, energy, N_BKELE_X );
        plasma.Advance( bmelec, grid, energy, N_BMELE_X );
        grid.DepositCharge( bkelec, Electron::charge, N_BKELE_X );
        grid.DepositCharge( bmelec, Electron::charge, N_BMELE_X );
        energy.tote(energy.pe() + energy.ke());
   }
} // End of Main
```

A.6 C++ Revised Scalar 1D/2D Main Program

This revised one-dimensional main program has an interface *identical* to the two-dimensional program. In two-dimensions, distribution class objects require additional parameters for initialization.

```
main()
{
Plasma plasma;
Electron elec[ NP ];
ParticleDist backelec( 1.0F, 0.0F, SYSLEN_X, N_BKELE_X );
ParticleDist beamelec( 0.5F, 5.0F, SYSLEN_X, N_BMELE_X );
Grid grid;
EnergyDiag energy;

  grid.Setup( energy );     // Initialize and construct grid

  backelec.Maxwellian( elec );
  beamelec.Maxwellian( &elec[N_BKELE_X] );

  grid.DepositCharge( elec, Electron::charge, NP );
  grid.AddIonDensity();

  for (int i = 0; i < N_STEPS; i++) {
       grid.CalcEField( energy );
       grid.InitChargeDensity();
       energy.ke(0.0);
       plasma.Advance( elec, grid, energy, NP );
       grid.DepositCharge( elec, Electron::charge, NP );
       energy.tote(energy.pe() + energy.ke());
  }
} // End of Main
```

The program interface is *identical* in two dimensions, however the distribution objects require additional parameters.

```
ParticleDist backelec( 1.0F, 1.0F, 0.0F, 0.0F, SYSLEN_X, SYSLEN_Y,
                       N_BKELE_X, N_BKELE_Y );
ParticleDist beamelec( 0.5F, 0.5F, 0.0F, 5.0F, SYSLEN_X, SYSLEN_Y,
                       N_BMELE_X, N_BMELE_Y );
```

A.7 C++ Scalar 3D Main Program

The main program for the scalar three-dimensional program is illustrated. Creation and initialization of the objects is illustrated, as well as the initialization and loop sections are shown.

```
main()
{
Plasma plasma;                                  // Collective Plasma
Vector< ChargedParticle > elec_pos( NP );    // Particle Vector Field
Vector< ChargedParticle > elec_vel( NP );
ParticleDist backgnd( N_BKELE_X, N_BKELE_Y, N_BKELE_Z, BKTHERMAL_VEL_X,
                      BKTHERMAL_VEL_Y, BKTHERMAL_VEL_Z, BKDRIFT_VEL_X,
                      BKDRIFT_VEL_Y, BKDRIFT_VEL_Z );
ParticleDist beam( N_BMELE_X, N_BMELE_Y, N_BMELE_Z, BMTHERMAL_VEL_X,
                   BMTHERMAL_VEL_Y, BMTHERMAL_VEL_Z, BMDRIFT_VEL_X,
                   BMDRIFT_VEL_Y, BMDRIFT_VEL_Z );
Field field( energy );      // Initialize and Create Collective Field
EnergyDiag energy;

  field.InitChargeDensity();

  plasma.UniformSpcMaxwellVelDist( elec_pos, elec_vel, field, backgnd );
  plasma.UniformSpcMaxwellVelDist( elec_pos, elec_vel, field, beam );

  field.ChargeDeposition( elec_pos, plasma, ChargedParticle::e_charge );
  field.AddBackgroundIonDensity();

  for ( int i = 0; i < N_STEPS; i++ ) {
        field.CalcEField( energy );
        field.SetBackgroundIonDensity();
        energy.ke( 0.0 );
        plasma.Advance( elec_pos, elec_vel, field, energy );
        field.ChargeDeposition( elec_pos, plasma,
                                ChargedParticle::e_charge );
        energy.tote( energy.pe() + energy.ke() );
  }
} // End Main
```

A.8 C++ Parallel 1D/2D/3D Main Program

The parallel program below was derived from the three-dimensional scalar program in appendix A.7.

```
main()
{
VPMachine vpm;                  // Parallel Machine Features
Vector< ChargedParticle > elec_pos( PTMAXNP );
Vector< ChargedParticle > elec_vel( PTMAXNP );
ParticleDist backgnd( N_BKELE_X, BKTHERMAL_VEL_X, BKDRIFT_VEL_X );
ParticleDist beam( N_BMELE_X, BMTHERMAL_VEL_X, BMDRIFT_VEL_X );
Plasma plasma;
EnergyDiag energy;
Field field( vpm, energy );

  vpm.ParInit();                // Initialize for Parallel Processing

  plasma.Partition( vpm );      // Particle/Field Partitioning
  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();

  for ( int i = 0; i < N_STEPS; i++ ) {
        field.CalcEField( vpm, energy );
        field.InitChargeDensity();
        energy.ke( 0.0 );
        plasma.Advance( elec_pos, elec_vel, field, energy, vpm );
        plasma.UpdateDistribution( elec_pos, elec_vel, vpm );
        field.ChargeDeposition( elec_pos, plasma,
                                ChargedParticle::e_charge );
        field.BackgroundIonDensity();
        energy.tote( energy.pe() + energy.ke() );
  }
} // End Main
```

The interface to the two-dimensional and three-dimensional programs is *identical*

to the one-dimensional version illustrated. The only visible difference involves the additional arguments required by the background and beam distribution objects. These objects are shown below.

A.8.1 Two-Dimensional Distribution Objects

```
ParticleDist backgnd( N_BKELE_X, N_BKELE_Y,
                      BKTHERMAL_VEL_X, BKTHERMAL_VEL_Y,
                      BKDRIFT_VEL_X, BKDRIFT_VEL_Y );

ParticleDist beam( N_BMELE_X, N_BMELE_Y,
                   BMTHERMAL_VEL_X, BMTHERMAL_VEL_Y,
                   BMDRIFT_VEL_X, BMDRIFT_VEL_Y );
```

A.8.2 Three-Dimensional Distribution Objects

```
ParticleDist backgnd( N_BKELE_X, N_BKELE_Y, N_BKELE_Z,
                  BKTHERMAL_VEL_X, BKTHERMAL_VEL_Y, BKTHERMAL_VEL_Z,
                  BKDRIFT_VEL_X, BKDRIFT_VEL_Y, BKDRIFT_VEL_Z );

ParticleDist beam( N_BMELE_X, N_BMELE_Y, N_BMELE_Z,
                  BMTHERMAL_VEL_X, BMTHERMAL_VEL_Y, BMTHERMAL_VEL_Z,
                  BMDRIFT_VEL_X, BMDRIFT_VEL_Y, BMDRIFT_VEL_Z );
```

A.9 Fortran 77 Parallel 1D Main Program

Much of the Fortran 77 parallel main program is shown with the specification of various data structures. This should be compared to the equivalent C++ program in appendix A.8.

```
c particles, charge/force arrays (real/complex)
      dimension part(idimp,npmax)
      dimension q(nxpmx), fx(nxpmx)
      dimension qc(kxp), fc(kxp)
c poisson, FFT tables
      dimension ffc(kxp)
      dimension mixup(kxp), sct(kxp)
c boundaries, particle transport buffers
      dimension edges(idps)
      dimension sbufl(idimp,nbmax), sbufr(idimp,nbmax)
      dimension rbufl(idimp,nbmax), rbufr(idimp,nbmax)
      dimension ihole(ntmax)
      dimension jsl(idps), jsr(idps)
c initialize for parallel processing
      call ppinit(idproc,nvp)
c calculate partition variables
      call dcomp1(edges,nxp,noff,nx,kstrt,nvp,idps)
c prepare fft tables
      isign = 0
      call pfft1r(qc,fc,isign,mixup,sct,indx,kstrt,kxp)
c calculate form factors
      call ppois1 (qc,fc,isign,ffc,ax,affp,we,nx,kstrt,kxp)
      if (npx.gt.0) call pistr1 (part,edges,npp,nps,vtx,zero,npx,nx,
     1idimp,npmax,idps)
c beam electrons
      nps = npp + 1
      if (npxb.gt.0) call pistr1(part,edges,npp,nps,vtdx,vdx,npxb,nx,
     idimp,npmax,idps)
c deposit charge for initial distribution
      call pdost1 (part,q,npp,noff,qme,idimp,npmax,nxpmx)
c add background ion density
      qi0 = -qme/affp
      do 170 j = 1, nxp
      q(j+1) = q(j+1) + qi0
  170 continue
```

```
  500 if (nloop.le.itime) go to 2000
c transform charge to fourier space
      isign = -1
c copy data from particle to field partition, and add up guard cells
      call cppfp1 (q,qc,isign,scr,kstrt,nvp,nxpmx,kxp,idps)
      call pfft1r(qc,fc,isign,mixup,sct,indx,kstrt,kxp)
c calculate force/charge in fourier space
      call ppois1 (qc,fc,isign,ffc,ax,affp,we,nx,kstrt,kxp)
c transform force/charge to real space
      isign = 1
      call pfft1r(fc,qc,isign,mixup,sct,indx,kstrt,kxp)
c copy data from field to particle partition, and copy to guard cells
      call cppfp1 (fx,fc,isign,scr,kstrt,nvp,nxpmx,kxp,idps)
c particle push and charge density update
c initialize charge density to zero
      do 1160 j = 1, nxp3
      q(j) = 0.
 1160 continue
      wke = 0.
c push particles
      call ppush1(part,fx,npp,noff,qtme,dt,wke,idimp,npmax,nxpmx)
c move particles into appropriate spatial regions
      call pmove1 (part,edges,npp,sbufr,sbufl,rbufr,rbufl,ihole,
     1jsr,jsl,jss,nx,kstrt,nvp,idimp,npmax,idps,nbmax,ntmax,ierr)
c deposit charge
      call pdost1 (part,q,npp,noff,qme,idimp,npmax,nxpmx)
c add background charge
      do 1180 j = 1, nxp
      q(j+1) = q(j+1) + qi0
 1180 continue
c energy diagnostic
      wt = we + wke
      if (kstrt.eq.1) write (6,993) we, wke, wt
      itime = itime + 1
      go to 500
 2000 continue
      stop
      end
```

A.10 C++ Parallel 3D Main Program (modified)

```
void main( int argc, char *argv[] )
{
// Parallel Machine Features
   VPMachine vpm;
// Initialize For Parallel Processing
   vpm.ParInit( NVPE );
// Particle Species Distribution Characteristics
   DistFunction backdf( N_BKELE_X, N_BKELE_Y, N_BKELE_Z,
                        SYSLENX, SYSLENY, SYSLENZ, BKTHERMAL_VEL_X,
                        BKTHERMAL_VEL_Y, BKTHERMAL_VEL_Z,
                        BKDRIFT_VEL_X, BKDRIFT_VEL_Y, BKDRIFT_VEL_Z );
   DistFunction beamdf( N_BMELE_X, N_BMELE_Y, N_BMELE_Z,
                        SYSLENX, SYSLENY, SYSLENZ, BMTHERMAL_VEL_X,
                        BMTHERMAL_VEL_Y, BMTHERMAL_VEL_Z,
                        BMDRIFT_VEL_X, BMDRIFT_VEL_Y, BMDRIFT_VEL_Z );
// The Particle Species
   Species< Particle3D > electrons( ECHARGE, ECHGMASS,
                                    (backdf.npxyz() + beamdf.npxyz()),
                                    SYSLENX, SYSLENY, SYSLENZ, vpm );
// The Scalar and Vector Fields
   ScalarField3D< float > cdensity( SYSLENX, SYSLENY, SYSLENZ, vpm );
   VectorField3D< float > efield( SYSLENX, SYSLENY, SYSLENZ, vpm );
// The Collective Field, Plasma Operations and Energy Diagnostic
   Fields3D fields( INDX, INDY, INDZ, vpm );
   Plasma plasma;
   EnergyDiag energy;
// ***************************************************************
// Begin Periodic 3D Electrostatic Particle Simulation Kernel Code
// ***************************************************************
   vpm.startclk();
// Prepare FFT Tables and Form Factors
   fields.SolvePrepare( E_HALFWIDTH_X, E_HALFWIDTH_Y, E_HALFWIDTH_Z,
                        electrons, vpm );
// Initialize Charge Density Profile
   cdensity = float( 0.0 );
// Perform Species Spatial and Velocity Distribution
   electrons.UniformDistribution( backdf, vpm );
   electrons.UniformDistribution( beamdf, vpm );
// Deposit Particle Charge to Field for Initial Distribution
   plasma.ChargeDeposition( electrons, cdensity );
```

```
// Add Background Ion Density
   const float QIO = ( -1.0 * electrons.qm *  electrons.npt ) /
                     ( cdensity.DIM_X() * cdensity.DIM_Y() *
                       cdensity.DIM_Z() );
   cdensity.add( QIO );
// ***************************************************************
// Begin Main Iteration Loop
// ***************************************************************
   for ( int i = 0; i < N_STEPS; i++ ) {
         fields.Solve( cdensity, efield, vpm );    // Solve Field
         plasma.pe( efield, energy );              // Field Energy
         cdensity.set( float( 0.0 ) );
         electrons.ke( 0.0 );
         plasma.Advance( electrons, efield, DT );  // Push Particles
         plasma.ke( electrons, energy );           // Kinetic Energy
         plasma.UpdateDistribution( electrons, vpm );
         plasma.ChargeDeposition( electrons, cdensity );
         cdensity.add( QIO );
         energy.tote();                            // Total Energy
   }
   vpm.stopclk();
   vpm.ParFinal();
} // End Main
```

A.11 Fortran 90 1D Main Program (initial)

Below we see the main program section of the Fortran 90 scalar program modeled after the C++ version in appendix A.6.

```
      program beps1k
      use Electron_m
      use ParticleDist_m
      use EnergyDiag_m
      use Plasma_m
      use Grid_m, only: Grid_Create, Grid_Setup,
     1Grid_InitChargeDensity, Grid_AddIonDensity,
     2Grid_CalcEField, Grid_DepositCharge
      use Timer_m
! object creation from derived types
      type (particle) elec(NP)
      type (species_dist) backelec
      type (species_dist) beamelec
      type (energy) energ
! create constructors for objects
      call ParticleDist_Create(backelec,vtx,zero,SYSLEN_X,N_BKELE_X)
      call ParticleDist_Create(beamelec,vtdx,vdx,SYSLEN_X,N_BMELE_X)
      call Grid_Create()
      call EnergyDiag_Create(energ)
! calculate form factors
      void = Grid_Setup()
! initialize density profile and velocity distribution
      call ParticleDist_Maxwellian (elec,backelec)
! beam electrons
      call ParticleDist_Maxwellian (elec(N_BKELE_X+1),beamelec)
! deposit charge for initial distribution
      call Grid_DepositCharge (elec,CHARGE,NP)
! add background ion density
      void =  Grid_AddIonDensity()
!
! * * * start main iteration loop * * *
!
      do 2000 i = 0, N_STEPS-1
      void = Grid_CalcEField(energ)
! particle push and charge density update
! initialize charge density to background
      void = Grid_InitChargeDensity()
```

```
      void = ke(energ,kin=0.)
! push particles
      call Plasma_Advance (elec,energ,NP)
! deposit charge
      call Grid_DepositCharge (elec,CHARGE,NP)
! energy diagnostic
      void = tote(energ,tot=(pe(energ)+ke(energ)))
      write (6,*) ' Time: ', i
      call EnergyDiag_Write(energ)
 2000 continue
      stop
      end
```

A.12 Fortran 90 1D Main Program (modified)

Below we see the main program section of the Fortran 90 modified scalar program. Objects are created and initialized by `module_create` routines.

```
      program beps1k
      use plasma_module
      type (distf1d) :: backdf, beamdf
      type (species1d) :: electrons
      type (fields1d) :: cdensity, efield
      type (energy) :: energ
      call distfunc_create(backdf,npx,nx,vtx,zero)
      call distfunc_create(beamdf,npxb,nx,vtdx,vdx)
      call species_create(electrons,qme,qbme,np)
      call fields_create(cdensity,nx)
      call fields_create(efield,nx)
      call energy_create(energ)
      call fields_solveprp(cdensity,indx,np,ax)
c initialize density profile and velocity distribution
      call fields_set (cdensity,zero)
c background electrons
      noff = 0
      if (npx.gt.0) call species_distr1(electrons,backdf,noff)
c beam electrons
      if (npxb.gt.0) call species_distr1(electrons,beamdf,npx)
c deposit charge for initial distribution
      call plasma_dpost1 (electrons,cdensity)
c add background ion density
      qi0 = -qme*float(np)/float(nx)
      call fields_add (cdensity,qi0)
c
c * * * start main iteration loop * * *
c
  500 if (nloop.le.itime) go to 2000
      write (6,991) itime
c calculate force/charge
      call fields_solve(cdensity,efield)
      call plasma_getpe(energ,efield)
c particle push and charge density update
c initialize charge density to background
      call fields_set (cdensity,qi0)
      wke = species_ke(electrons,wks=0.)
```

```
c push particles
      call plasma_push1 (electrons,efield,dt)
c deposit charge
      call plasma_dpost1 (electrons,cdensity)
      call plasma_getke(energ,electrons)
c energy diagnostic
      call plasma_gette(energ)
      itime = itime + 1
      go to 500
 2000 continue
c
c * * * end main iteration loop * * *
c
c unallocate data
      call fields_destroy(efield)
      call fields_destroy(cdensity)
      call species_destroy(electrons)
      stop
      end
```

A.13 Fortran 90 2D Main Program

Below we see the main program section of the Fortran 90 two-dimensional scalar program. This version should be compared to the one-dimensional version of appendix A.12.

```
      program beps2k
      use plasma_module
      type (distf2d) :: backdf, beamdf
      type (species2d) :: electrons
      type (sfields2d) :: cdensity
      type (vfields2d) :: efield
      type (energy) :: energ
      call distfunc_create(backdf,npx,npy,nx,ny,vtx,vty,zero,zero)
      call distfunc_create(beamdf,npxb,npyb,nx,ny,vtdx,vtdy,vdx,vdy)
      call species_create(electrons,qme,qbme,np)
      call fields_create(cdensity,nx,ny,nxv)
      call fields_create(efield,nx,ny,nxv)
      call energy_create(energ)
c prepare fft tables and form factors
      call fields_solveprp(cdensity,indx,indy,np,ax,ay)
c initialize density profile and velocity distribution
      call sfields_set(cdensity,zero)
c background electrons
      noff = 0
      if (npxy.gt.0) call species_distr2(electrons,backdf,noff)
c beam electrons
      if (npxyb.gt.0) call species_distr2(electrons,beamdf,npxy)
c deposit charge for initial distribution
      call plasma_dpost2 (electrons,cdensity)
c add background ion density
      qi0 = -qme*float(np)/float(nx*ny)
      call sfields_add(cdensity,qi0)
c
c * * * start main iteration loop * * *
c
  500 if (nloop.le.itime) go to 2000
      write (6,991) itime
c calculate force/charge
      call fields_solve(cdensity,efield)
      call plasma_getpe(energ,efield)
```

```
c particle push and charge density update
c initialize charge density to background
      call sfields_set(cdensity,qi0)
      wke = species_ke(electrons,wks=0.)
c push particles
      call plasma_push2 (electrons,efield,dt)
c deposit charge
      call plasma_dpost2 (electrons,cdensity)
      call plasma_getke(energ,electrons)
c energy diagnostic
      call energy_gette(energ)
      itime = itime + 1
      go to 500
 2000 continue
c
c * * * end main iteration loop * * *
c
c unallocate data
      call fields_solvetrm
      call fields_destroy(efield)
      call fields_destroy(cdensity)
      call species_destroy(electrons)
      stop
      end
```

A.14 Fortran 90 Parallel 2D Main Program

Below we see the main program section of the Fortran 90 two-dimensional parallel program. This version should be compared to the two-dimensional scalar version of appendix A.13, as well as the C++ parallel program of appendix A.10.

```
      program beps2k
      use partition_module
      use plasma_module
      type (distf2d) :: backdf, beamdf
      type (species2d) :: electrons
      type (sfields2d) :: cdensity
      type (vfields2d) :: efield
      type (energy) :: energ
      type (slab) :: edges
      call MPI_INIT(ierror)
      call ppinit(nvp)
      call distfunc_create(backdf,npx,npy,nx,ny,vtx,vty,zero,zero)
      call distfunc_create(beamdf,npxb,npyb,nx,ny,vtdx,vtdy,vdx,vdy)
      call species_create(electrons,qme,qbme,np,nvp)
      call fields_create(cdensity,nx,ny,nxv,nyv,nvp)
      call fields_create(efield,nx,ny,nxv,nyv,nvp)
      call energy_create(energ)
! calculate partition variables
      call dcomp2(edges,ny)
! prepare fft tables and form factors
      call fields_solveprp(cdensity,indx,indy,np,ax,ay,nvp)
! initialize density profile and velocity distribution
      call sfields_set(cdensity,zero)
! background electrons
      noff = 1
      if (npxy.gt.0) call species_distr2(electrons,edges,backdf,noff)
! beam electrons
      nps = electrons%npp + 1
      if (npxyb.gt.0) call species_distr2(electrons,edges,beamdf,nps)
! deposit charge for initial distribution
      call plasma_dpost2 (electrons,cdensity,edges)
! add background ion density
      qi0 = -qme*float(np)/float(nx*ny)
      call sfields_add(cdensity,edges,qi0)
!
```

```
! * * * start main iteration loop * * *
!
  500 if (nloop.le.itime) go to 2000
      if (kstrt.eq.1) write (6,991) itime
! calculate force/charge
      call fields_solve(cdensity,efield)
      call plasma_getpe(energ,efield)
! particle push and charge density update
! initialize charge density to zero
      call sfields_set(cdensity,zero)
! initialize charge density to background
      wke = species_ke(electrons,wks=0.)
! push particles
      call plasma_push2 (electrons,efield,edges,dt)
! move particles into appropriate spatial regions
      call plasma_pmove2 (electrons,edges,ny,nbmax,ntmax)
! deposit charge
      call plasma_dpost2 (electrons,cdensity,edges)
! add background ion density
      call sfields_add(cdensity,edges,qi0)
      call plasma_getke(energ,electrons)
! energy diagnostic
      if (kstrt.eq.1) then
         call energy_gette(energ)
      endif
      itime = itime + 1
      go to 500
 2000 continue
!
! * * * end main iteration loop * * *
!
! unallocate data
      call fields_solvetrm
      call fields_destroy(efield)
      call fields_destroy(cdensity)
      call species_destroy(electrons)
      call MPI_FINALIZE(ierror)
      stop
      end
```

A.15 C++ Parallel 3D Free-Expansion Main Program Sketch

Below we see the main program section of the C++ three-dimensional parallel program with load balancing. This version should be compared to the C++ parallel program of appendix A.10 on page 201 since only the modifications are sketched.

```
void main( int argc, char *argv[] )
{
// The Particle Species
   Species< Particle3D > electrons( ECHARGE, ECHGMASS,
                                    (backdf.npxyz() + beamdf.npxyz()),
                                    SYSLENX, SYSLENY, SYSLENZ, vpm );
   Species< Particle3D > ions( ICHARGE, ICHGMASS,
                               (backidf.npxyz() + beamidf.npxyz()),
                               SYSLENX, SYSLENY, SYSLENZ, vpm );
// ****************************************************************
// Begin Main Iteration Loop
// ****************************************************************
   for ( int i = 0; i < N_STEPS; i++ ) {
         fields.Solve( cdensity, efield, vpm );
         plasma.pe( efield, energy );
         cdensity.set( float( 0.0 ) );
         electrons.ke( 0.0 );
         plasma.Advance( electrons, efield, DT );
         plasma.Advance( ions, efield, DT );
         plasma.ke( electrons, energy );
         plasma.UpdateDistribution( electrons, vpm );
         plasma.UpdateDistribution( ions, vpm );

         // Load balance the particles (if necessary)
         plasma.BalanceDistributionRB( electrons, ions,
                                       cdensity, efield, vpm );

         plasma.ChargeDeposition( electrons, cdensity );
         plasma.ChargeDeposition( ions, cdensity );
         energy.tote();
   }
} // End Main
```

A.16 C++ Parallel 3D Gravitation Main Program Sketch

Below we see the main program section of the C++ three-dimensional parallel program with load balancing. This version should be compared to the C++ parallel program of appendix A.15 on page 211 since only the modifications are sketched.

```
void main( int argc, char *argv[] )
{
// The Particle Species
   Species< Particle3D > electrons( ECHARGE, ECHGMASS,
                                    (backdf.npxyz() + beamdf.npxyz()),
                                    SYSLENX, SYSLENY, SYSLENZ, vpm );
// ****************************************************************
// Begin Main Iteration Loop
// ****************************************************************
   for ( int i = 0; i < N_STEPS; i++ ) {
         fields.Solve( cdensity, efield, vpm );
         plasma.pe( efield, energy );
         cdensity.set( float( 0.0 ) );
         electrons.ke( 0.0 );
         plasma.Advance( electrons, efield, DT );
         plasma.ke( electrons, energy );
         plasma.UpdateDistribution( electrons, vpm );

         // Load balance the particles (if necessary)
         plasma.BalanceDistributionRB( electrons,
                                       cdensity, efield, vpm );

         plasma.ChargeDeposition( electrons, cdensity );
         cdensity.add( QIO );     // Background Density
         energy.tote();
   }
} // End Main
```

Index