UCLA Parallel PIC Framework

Viktor K. Decyk[1,2], Charles D. Norton[2]

[1]*Department of Physics and Astronomy*
University of California, Los Angeles
Los Angeles, California 90095-1547

[2]J*et Propulsion Laboratory*
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109-8099

Abstract

The UCLA Parallel PIC Framework (UPIC) has been developed to provide trusted components for the rapid construction of new, parallel Particle-in-Cell (PIC) codes. The Framework uses object-based ideas in Fortran95, and is designed to provide support for various kinds of PIC codes on various kinds of hardware. The focus is on student programmers. The Framework supports multiple numerical methods, different physics approximations, different numerical optimizations and implementations for different hardware. It is designed with "defensive" programming in mind, meaning that it contains many error checks and debugging helps. Above all, it is designed to hide the complexity of parallel processing. It is currently being used in a number of new Parallel PIC codes.

Corresponding author: Viktor K. Decyk, UCLA Department of Physics and Astronomy, Los Angeles, CA 90095-1547, USA. Tel. (310) 206-0371, FAX 310-825-4057, email: decyk@physics.ucla.edu.

I. Frameworks

Frameworks are an emerging technology for code reuse in development of complex software systems [1,2]. They are generally based on an object-oriented design, with interacting classes that can be specialized to produce custom applications. They have been largely used in developing business software, with only a few examples in scientific computing [3]. While frameworks are commonly implemented in an object-oriented language, this is not a requirement.

Although there are many definitions of a framework, the working definition we will adopt is that a framework is a unified environment containing all the components needed for writing code for a specific problem domain. Its goal is the rapid construction of new codes by reusing trusted modules. The framework under development here is designed for student programmers. Although it can be used to write a large "mega-code," it is designed to provide "Lego" pieces for rapid construction of new codes. Our framework will support multiple numerical methods, different physics approximations, different numerical optimizations and implementations for different hardware. It is designed with "defensive" programming in mind, meaning that it contains many error checks and debugging helps. Above all, it is designed to hide the complexity of parallel processing.

Why is such a framework important? Students doing Ph. D. research in computational physics typically inherit a code from another researcher, and modify it to solve some new problem. This process often takes six months or a year. There are several reasons for this long time. First is that physics students typically have only minimal training in programming, and usually none at all in software engineering. They are generally focused only on making new scientific discoveries: there are no rewards for students to spend the extra time and training needed to write software that could be easily reused by others. As a result, when a new student inherits a code, much of the code must be understood before one can make changes safely, even

though the student typically wants to modify only that piece which is crucial to his new problem.

Our solution to this dilemma is to write powerful high level classes that can easily be reused for those parts of the code which the students do not intend to modify. These high level classes provide simple interfaces that encapsulate the implementation details of a large block of code. They typically contain hidden pointers to the structures needed to perform various functions, such as a field class which hides tables needed to perform a Fast Fourier Transform (FFT). There are many examples of such high level classes in commercial software development, such as the Cocoa environment (derived from NextStep) on the Macintosh, which allows one to add word processing capabilities to an application in less than 50 lines of code.

Invariably, students need to modify some high level class for their research needs. To simplify this process, we find that it is useful to build high-level classes from middle-layer helper classes. These helper classes contain descriptors of data, but not the data themselves. For example, for a parallel program, a helper class can contain information on how an array is laid out on different processors, and it is desirable that related arrays have the same layout to minimize communication. These helper classes are thus a primary means of hiding parallel processing details. They can also contain flags that allow polymorphism in non-object-oriented languages. For example, for a module which solves Poisson's equation, the helper class can contain a flag to indicate what kind of solver to use, hiding this detail, and allowing the same simple interface to be used regardless of the method of solution. Finally, by separating the descriptors of data from the data itself, it is easier to incorporate new helper classes into already existing high-level classes. It is a well-known problem in object-oriented design that it is difficult to reuse a class if there is already another one with similar, but not identical, functionality. These helper classes help solve this problem, since only a smaller helper class needs to be "inherited" to obtain a new

functionality, while the array of the original class is reused.

II. Framework Design for Parallel Plasma PIC

We are implementating such a framework for parallel plasma Particle-in-Cell (PIC) simulations. PIC codes model plasmas as particles, integrating the trajectories of millions of particles interacting self-consistently via electromagnetic fields. Particles do not interact directly with each other, but rather with electromagnetic fields they produce, which are calculated on a grid. This reduces the computational complexity to order N, where N is the number of particles. The basic algorithm is straightforward. From the particles' coordinates, one can calculate an approximate charge and current density in space by accumulating these quantities on a grid by inverse interpolation. Typically, linear interpolation is used, so that in three dimensions, a particle would deposit these quantities on the 8 grid points nearest to its location. The charge and current densities are used as source terms in solving Maxwell's equations to obtain the electric and magnetic fields on the same grid. If the source terms are known, this is a set of linear partial differential equations which are generally straightforward to solve and not time-consuming. Finally, the particle coordinates are updated using Newton's law by interpolating the electromagnetic fields to the particles' location. There are a wide variety of PIC codes. Some codes solve only a subset of Maxwell's equations, others include relativity, still others include other forces in the calculation, such as collisions. PIC codes have been extensively used in all areas of plasma physics since the early 1960s. There are several textbooks on this topic [4,5].

PIC codes make the fewest approximations of any physics model, but they are also the most computationally demanding. They are extensively used when more approximate models, such as fluid models, fail. They are also used to test and determine the realm of validity of other new reduced description models. Sometimes they are used even if fluid models would work, just because of their simplicity. The UCLA plasma simulation group, started by the late John

4

Dawson, has been a major developer of PIC codes for nearly 30 years. We have pioneered the development of algorithms for parallel PIC codes [6,7], as illustrated in Fig. 1. Our group has a large legacy of trusted subroutines from many codes, largely written in Fortran77 and Fortran90. Therefore our framework implementation is based on Fortran95.

Fortran95 is a powerful high level language designed for mathematical modeling, and physics students learn it quite easily. It is a safe language that gives high performance, with many features attractive to scientists who want to spend a minimum amount of time programming, such as leak-proof dynamic memory. However, as we shall see, our framework is designed in layers so that other languages can be used. The methodology for how to implement object-oriented concepts in Fortran95 is based on ideas previous developed by Decyk, Norton, and Szymanski [8-9].

The current framework is designed to support two and three dimensional parallel codes. It uses trusted legacy Fortran77 subroutines as the lowest layer. These legacy subroutines are well debugged and provide most of the basic functionality, but are not intended to be modified greatly. The most CPU time-consuming parts of a PIC code are the particle push and charge deposit. These subroutines have been very carefully written to provide the highest performance possible [10].

It is important for research codes to support multiple models and numerical schemes. For example, linear interpolation is generally used for PIC codes, but the easiest way to verify that linear interpolation was sufficiently accurate is to run a few cases with quadratic interpolation and see if the results changed. Different algorithms are used with different hardware. For example, a different scheme is used to deposit charge on a vector machine than on a RISC processor. The legacy layer has available subroutines to support linear and quadratic interpolation, both message-passing (MPI) and shared memory parallel programming (pthreads,

OpenMP), and can support both RISC and vector architectures. However, using these features at the Fortran77 layer is complex and error prone, although the performance and functionality are very good.

Since Fortran77 is a relatively simple language, it would be straightforward to translate some of these legacy subroutines into another language such as C, without impacting the framework. In fact, there exist automatic tools to perform such translations.

III. Implementation of Middle Layer Helper classes for Parallel PIC

The middle layer is written in Fortran95 and encapsulates this legacy code with simple interfaces and provides additional safety checks. Most of the current development has been for the 2D code, because it is an simpler environment for developing new approaches. There are currently six helper classes implemented for the 2D Framework. They are:

1. A uniformly partitioned field class, which contains layout information and provides transpose and guard cell methods for arrays where each processor contains the same size subarray. Transpose methods are particularly useful for implementing any algorithm where one calculates on all the rows then all the columns separately, such as a 2D FFT.

2. A non-uniformly partitioned field class, which contains layout information and guard cell methods for arrays where each processors' subarray may be different (as shown in Fig. 1).

3. An FFT class, which contains information about FFT tables and provides parallel FFT methods.

4. A Poisson solver class, which contains tables needed by various parallel spectral-based field

solvers and provides Poisson and Maxwell solvers. Three types of boundary conditions are currently implemented, periodic, Dirichlet, and mixed periodic/Direchlet. Vacuum boundary conditions are planned for the future.

5. A particle distribution class, which contains information about particle distribution functions.

6. A particle class, which contains information describing particles and includes methods for depositing charge, pushing particles, and managing particles on parallel computers. Currently both magnetized and unmagnetized particles are supported, with relativity as an option, with many different implementations.

In addition, there are utility classes that provide: a) parallel processing services such as timers and I/O, b) parallel error handling including tracing subroutine calls and dynamic memory, as well as testing pre-conditions and post-conditions, and c) a limited set of parallel graphics.

The middle layer primarily provides a much safer and simpler interface to the complex Fortran77 legacy subroutines by encapsulating many details, such as data layouts on parallel machines. It also provides polymorphism, and as a run-time option, verifies that required pre- and post-conditions are satisfied when calling the legacy code. It does not currently perform any demanding calculations, so high performance is not required. There is nothing crucial in the framework which is platform dependent. This middle layer could be rewritten in another object-based or object-oriented language that can call Fortran77, such as C++, or a scientific programming environment such as IDL, MATLAB, or Mathematica, without impacting the lower layers.

7

To illustrate this , below is what a 2D Parallel FFT call looks like in the original legacy

Fortran77 code:

```
call PFFT2RX2(fxye,fxyt,isign,ntpose,mixup,sct,indx,indy,
        kstrt,nxeh,nyv,kxp,kyp,nypmx,jblok,kblok,nxhy,nxyh)
```

This is what it looks like with the middle layer helper classes, where various arguments which

logically belong together have been encapsulated in useful abstractions:

```
call fft(ftable,rspace,fxye,kspace,fxyt,isign)
```

In this case, ftable is an object of the fft class, rspace and kspace are objects of the uniformly

partitioned field class, and fxye, and fxyt are the input and output arrays. Another example is

the parallel particle manager that checks and moves particles between processors. In Fortran77

this subroutine looks like:

```
call PMOVE2(part,edges,npp,sbufr,sbufl,rbufr,rbufl,ihole,
        jsr,jsl,jss,ny,kstrt,nvp,idimp,npmax,nblok,idps,nbmax,
        ntmax,ierr)
```

with the middle layer helper classes, it is much simpler:

```
call pmove(epart,part,nspace)
```

Where epart is an object of the particle class, nspace is an object of the non-uniformly partitioned

field class, and part is the particle array. Since Fortran95 does not implement traditional object-

oriented syntax, by convention, the first argument in the subroutine is the object of the method.

With helper classes, the objects do not contain the actual data. The data is an argument.

IV. Implementation of High Level classes for Parallel PIC

Three high-level classes has been written. They are:

 1. A scalar fields class, which can be FFTed or be the solution of a Poisson equation. The data is

contained within the object. Because the FFT transposes the data, the object keeps track of its

current state. For some types of boundary conditions, the grid must be doubled in size. This feature is internal to the object and hidden from the user. A charge density or potential are examples of such objects.

2. A vector fields class, which can be FFTed or be the solution of a Poisson equation. A current density or electric fields are examples of such objects.

3. A species class, which contains particle co-ordinates in an array part, as well as a particle helper object which describes properties of a particle. There are three main methods in the class, depositing charge and current and pushing particles. The internal helper object determines what algorithms are used in each case.

In general, high level classes are designed to have various properties to enable it to respond to "messages" to perform some action. For example, in order for an object to be FFTable, it must encapsulate within itself pointers to appropriate fft tables. Typical properties include being writable for output or displayable on the screen. The user then only needs to be concerned with constructing the object appropriately, or adding or deleting properties to an object already created. Once constructed, the use is trivial. For example, to FFT a scalar field object, one merely calls:

```
call fftkr(fxy)
```

Similarly, to advance electrons in time, one merely calls:

```
call push(electrons,efield,bfield,energy)
```

where efield and bfield are vector field objects, and energy is a real number. The particle manager is called internally by the push, so its function is hidden from the user.

In addition to these classes, there is also a plasma class, which encapsulates the entire plasma.

This allows the main simulaton program to be written in only a few lines of code:

```
      use plasma2d_class
      integer :: done = 0
      type (plasma2d) :: plasma
!
      call new_plasma2d(plasma)
      do while (done >= 0)
         done = update_plasma(plasma)
      enddo
!
      call del_plasma2d(plasma)
```

With this class, one could incorporate the entire plasma simulation code very simply into another code. One can also easily rewrite this main code in another language, such as Java, so that the plasma simulation code could be controlled from another environment, such as a web browser.

V. Use of Framework for new Parallel PIC codes

The framework currently supports various electromagnetic models. It can be electrostatic (coulomb interaction only), magnetized electrostatic (coulomb interaction with external magnetic fields), Darwin (Maxwell's equation without radiation), and fully electromagnetic (Maxwell's equations), with various boundary conditions. Relativity is supported. Energy conservation is outstanding, with energy conservation of about 1 part in 10^5 or 10^6 for 4000 time steps achievable. The 3D code been run on various parallel machines and scales well up to 2048 processors and 12 billion particles. Some performance results are in Table I.

The framework is currently being used by a number of new PIC codes. One such code is QuickPIC [11-12], a quasi-static code for studying plasma-based accelerators. It consists of a coupled 3D and 2D PIC code. Another code is QPIC [13], a 2D quantum PIC code using a semi-classical scheme based on integrating classical Feynman paths. A third code is a new version of BEPS [14], a code used in teaching plasma physics. Recently, development of a cosmology code has been undertaken.

Acknowledgment

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns {Addison-Wesley, Boston. 1995].

[2] Mohamed E. Fayad, Douglas C. Schmidt and Ralph E. Johnson, Building Application Frameworks [John Wiley & Sons, New York, 1999].

[3] K. Shadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai, "Challenges in Computer Architecture Evaluation," IEEE Computer, Aug. 2003, p. 30.

[4] R. W. Hockney and J. E. Eastwood, Computer Simulation Using Particles [McGraw-Hill, New York, 1981].

[5] C. K. Birdsall and A. B. Langdon, Plasma Physics via Computer Simulation [McGraw-Hill, New York, 1985].

[6] P. C. Liewer and V. K. Decyk, "A General Concurrent Algorithm for Plasma Particle-in-Cell Codes," J. Computational Phys. 85, 302 (1989).

[7] V. K. Decyk, "Skeleton PIC Codes for Parallel Computers," Computer Physics Communications 87, 87 (1995).

[8] V. K. Decyk, C. D. Norton, and B. K. Szymanski, "How to Express C++ Concepts in Fortran90," Scientific Programming, 6, 363 (1997).

[9] V. K. Decyk, C. D. Norton, and B. K. Szymanski, "How to support inheritance and run-time polymorphism in Fortran 90", Computer Physics Communications 115, 9, (1998).

[10] V. K. Decyk, S. R. Karmesin, A. de Boer, and P. C. Liewer, "Optimization of Particle-in-Cell codes on RISC processors," Computers in Physics 10, 290 (1996).

[11] C. Huang, V. Decyk, S. Wang, E.S.Dodd, C. Ren, W. B. Mori, "A Parallel Particle-in-Cell Code for Efficiently Modeling Plasma Wakefield Accleration: QuickPIC," Proc. of the 18th Annual Review of Progress in Applied Computational Electromagnetics, Monterey, CA, March, 2002, p.557.

[12] G. Rumolo, A. Z. Ghalam, T. Katsouleas, C. K. Huang, V. Decyk, C. Ren, W. Mori, F. Zimmerman, and F. Ruggiero, "Electron cloud effects on beam evolution in a circular accelerator," to be published in Physical Review STAB, 2003.

[12] Dean Dauger, "Semiclassical Modeling of Quantum-Mechanical Multiparticle Systems using

Parallel Particle-in-Cell Methods." Ph. D. Dissertation, UCLA, Los Angeles, California [2001].

[13] Decyk, V.K. and Slottow, J.E., "Supercomputers in the Classroom," Computers in Physics, vol. 3, No. 2, 1989, p. 50.

Tables

Table I: Large 3D Electrostatic Particle Benchmarks

```
---------------------------------------------
The following are times for a 3D particle simulation, using
127,401,984 particles and a 256x32x512 mesh.  Push Time is the
time to update one particle's position and deposit its charge, for
one time step.  Speed is calculated from push time by using 125
Flops/particle/time-step.  The code uses linear interpolation.
---------------------------------------------
Computer                        Push Time        Speed

---------------------------------------------
IBM SP3/375, w/MPI, 1024 proc:    1.4 ns.       89.3 GFlop
IBM SP3/375, w/MPI, 512 proc:     2.0 ns.       62.5 GFlop
IBM SP3/375, w/MPI, 256 proc:     4.2 ns.       29.8 GFlop
IBM SP3/375, w/MPI, 128 proc:     7.4 ns.       16.9 GFlop
IBM SP3/375, w/MPI, 64 proc:      15 nsec.       8.3 GFlop
IBM SP3/375, w/MPI, 32 proc:      28 nsec.       4.5 GFlop
IBM SP3/375, w/MPI, 16 proc:      56 nsec.       2.2 GFlop
---------------------------------------------
```
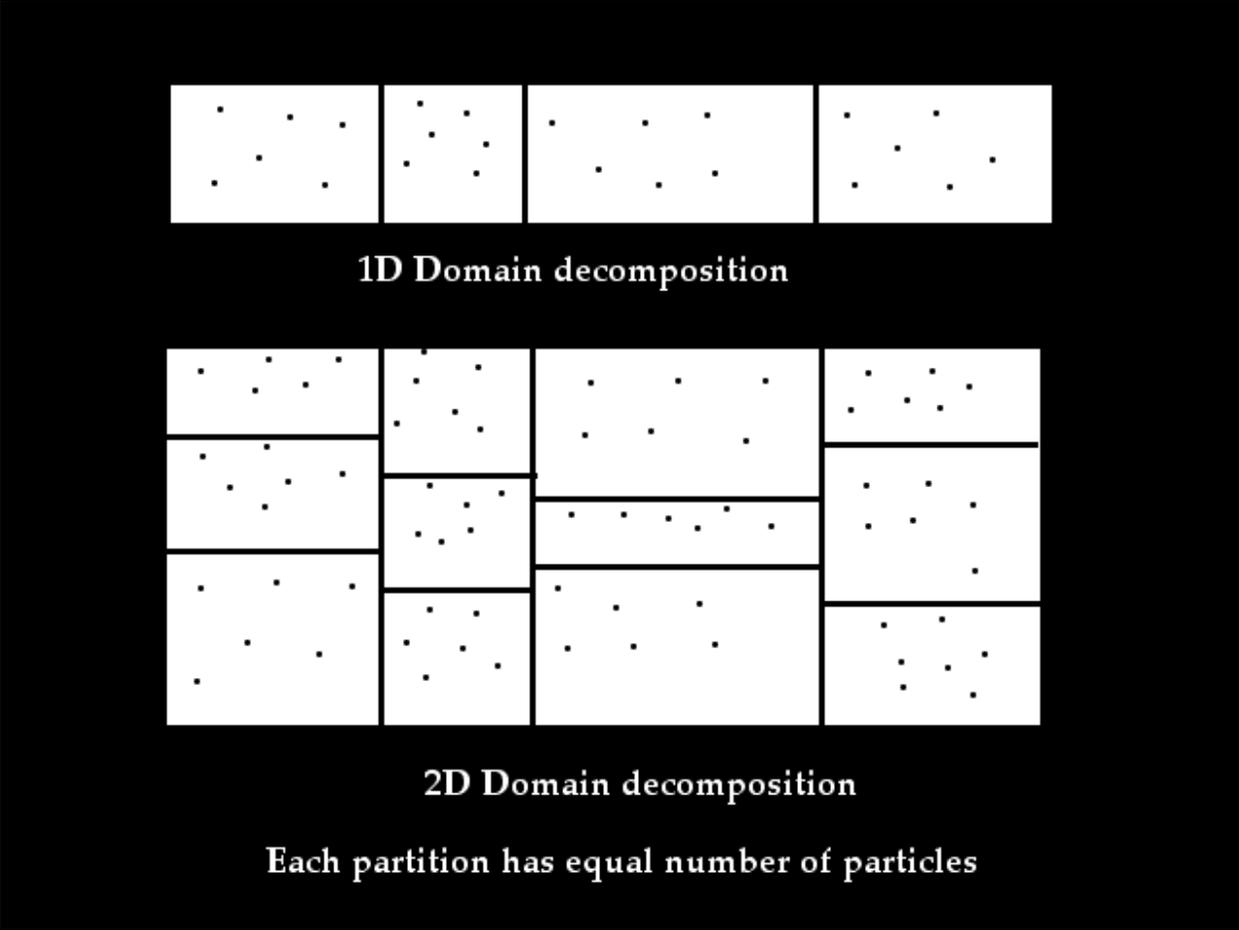
Figure 1: Diagram shows how space can be partitioned among different processors in a parallel PIC code.