Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Computational Physics 196 (2004) 448–473
JOURNAL OF COMPUTATIONAL PHYSICS
www.elsevier.com/locate/jcp

VORPAL: a versatile plasma simulation code

Chet Nieter *, John R. Cary

Center of Integrated Plasma Studies and Department of Physics, University of Colorado, Campus Box 390 Boulder, CO 80309-0390, USA

Received 22 January 2003; received in revised form 6 November 2003; accepted 9 November 2003

Abstract

VORPAL is a new plasma simulation code designed for maximum flexibility through use of advance C++ techniques. Through use of inheritance, VORPAL incorporates multiple models for the plasma and electromagnetic fields. The plasma models include both particle-in-cell and fluid models. Through C++ meta-template programming a single code can be used to simulate one-, two-, or three-dimensional systems with no loss of performance. VORPAL can also be run in either serial or parallel, with the latter using a general domain decomposition. A new fluid algorithm that allows for regions of zero density was developed and incorporated into the code. VORPAL simulation results for the generation of laser wake fields through laser–plasma interaction are presented.

PACS: 52.65.Kj; 52.65.Rr; 52.65.Ww; 52.38.Kd

Keywords: Plasma physics; Object-oriented programming; Hybrid simulations; Laser–plasma interactions

1. Introduction

From the earliest days of computers, computation has played a major role in plasma research. Since then, plasma computation [1,2] has made great strides, with advances in algorithms, simulation of reduced models and models for slow plasma phenomena, and usability. The goal of the present paper is to describe a new plasma simulation code, VORPAL, that has been designed from the outset for greater versatility, through use of Object Oriented methods along with use of advanced features in the C++ programming language. Our code incorporates many of the ideas of the OOPIC code [3,4], a Particle-In-Cell (PIC) code that can easily incorporate different types of electromagnetic (EM) fields, particle dynamics, particle boundaries, field boundaries, etc. Many of these ideas are also present in the Fortran 90 codes OSIRIS [5] and 3D codes based off the GCPIC algorithm [6].

Our goal for VORPAL was to develop a new framework of classes that would take the maximal advantage of Object Oriented methods without sacrificing performance. As part of this effort, we added

* Corresponding author.
E-mail addresses: nieter@colorado.edu (C. Nieter), cary@colorado.edu (J.R. Cary).

0021-9991/$ - see front matter
doi:10.1016/j.jcp.2003.11.004

further generalizations to the VORPAL framework that are not available in other object-oriented plasma codes. The first of which is VORPAL can be run in any number of dimensions and the number of dimensions can be set at run time. It has a flexible domain decomposition algorithm which allows for any 3D domain decomposition that is comprised of a collection of slabs. The framework allows both fluid and particle representations of the plasma and these representations can be used independently or together to perform hybrid simulations. Since we are interested in the interactions of high-intensity laser pulses with plasmas, we developed a new fluid algorithm which can model regions of zero density so we can deal with the plasma blow outs. Our particle model uses policy class methods to support multiple types of particle dynamics easily and efficiently. To make sure that these new features are implemented lower in the class hierarchy so that any new models added to the code would automatically inherit them, we choose to construct the low level physics objects and the interfaces in the framework ourselves rather than building VORPAL on top of an existing framework. However, we recognize that one should not reinvent the wheel unless necessary. So we make extensive use of math objects from the OptSolve++ libraries from Tech-X corporation [7] and container objects from the Standard Template Library [8].

The VORPAL framework is organized in a manner that reflects its object-oriented design. One package (subdirectory and associated library) called `vpbase` contains all of the basic interfaces such as the interface for the electromagnetic field that has methods to update the field and obtain the value of the field at a point. All the objects in the code interact through the interfaces defined in `vpbase`. The next layer of packages contain classes which implement the behaviors for the methods in the interface classes, including those for the EM fields, particles, fluids, messaging, and I/O. Finally, VORPAL has a package for the domain, which combines objects to create a simulation. In this way, VORPAL has a package hierarchy without mutual dependencies. This simplifies reuse of the code libraries.

In laser–plasma interactions the great disparity in scales leads to large computational requirements. For example, with a laser pulse commonly containing 50 laser oscillations and the need for simulating regions that are multiple pulse lengths in each direction, it can be necessary to have simulations of 10,000 cells along the direction of propagation and 500 transverse. This results in 5×10^6 cells for a 2D simulation and 2.5×10^9 cells in 3D. Thus, to obtain results rapidly, one often does most simulations in 2D, with rarer 3D simulations, often with scaled parameters. In the usual methodology, one writes separate 2D and 3D codes, or one has only a single 3D code, which for 2D runs is restricted to a small number of cells (2–4) in the symmetry direction. Unfortunately, this is computationally more intensive than a 2D simulation.

Our solution to these problems was to build an arbitrary-dimensional code that depends on recursion; the nested loops of a multiple-dimensional field update are constructed through a function calling itself for each loop, with the recursion ending at the dimensionality. However, function calls can be computationally expensive, so it is desirable to have this recursion inlined. However, this cannot be done for recursive calls of the same function. To effect this we use the template mechanism of C++. This allows us to have each of the nested function calls be to a function of a different template parameter, which becomes a different object-code function, and so inlining is possible. Thus VORPAL has arbitrary dimensionality without the loss of efficiency associated with such recursion, providing the same performance that would be achieved through the use of nested loops. This simplifies maintenance, as one no longer needs to maintain a separate code for each dimensionality.

VORPAL achieves parallelism through domain decomposition. The VORPAL decomposition is more general in that domain boundaries need not line up in any particular way. Instead, each domain is a slab, and so the full simulation region is then any region that can be created through combining slabs. The messaging, a bit more complex than usual, is determined by finding the intersections of each domain with the extended region (that includes guard cells) of all neighboring domains.

Due to excessive virtual function calls, traditional inheritance is not an efficient way of implementing multiple particle models into a code. VORPAL has a single particle class, whose dynamics are determined by a separate class, referred to as the policy class. The update methods in the policy class are not virtual and

therefore can be inlined away. Finally we note that in VORPAL, plasmas can also be simulated by fluid representations. For this purpose, we have developed a simple algorithm applicable to cold plasma that advects the velocity field, then updates the density through a flux corrected transport scheme. Since we are advecting the fluid velocity rather than updating the fluid momentum, we can model regions of zero density.

VORPAL has now matured and is being used to study a variety of plasma phenomenon. The original target for VORPAL was to study Laser Wake Field Acceleration (LWFA) [9]. Firing an intense laser pulse into a plasma with pulse length near the plasma wavelength can generate plasma oscillations behind the pulse having electric fields on the order of 100 GeV/m. With its current capabilities VORPAL has been used to study the generation of laser wake fields and the optical injection of electron beams into these wake fields [10]. The above-mentioned fluid algorithm allows hybrid simulations of optical injection to be done where the beam is modeled with particles, and the wake field is modeled with a fluid. For such simulations, the laser propagates in from vacuum, and the ponderomotive force from the laser pulse can completely blow out all the plasma from the region of the laser pulse, so we had to develop a fluid algorithm that can deal with zero density regions to use in these hybrid simulations. Recently, VORPAL has been used to to study electron Bernstein heating in fusion physics. Preliminary simulations have been done and we are planning on adding new physics models to assist in this endeavor.

The rest of the paper is organized as follows. In Section 2, we discuss the life cycle of objects in the code and the organization of the classes in the code framework. In Section 3, we explain our method of arbitrary-dimensional coding that allows us to specify the dimension of the simulation at run time. Section 4 details the method we used to provide a general 3D domain decomposition. All of the available physics models and their implementations are discussed in Section 5. Tests of the code to show it is producing correct physics are given in Section 6. In Section 7, we discuss the results of a hybrid fluid-PIC simulation of LWFA made possible by VORPAL's object-oriented features. Finally in Section 8, we summarize our work and discuss some the future plans and possible new features of VORPAL.

2. Architecture and code techniques

The goal of a plasma simulation code is to evolve a numerical representation of a plasma in an electromagnetic field. Thus, the basic steps are: (1) creation and initialization of the objects that comprise the plasma, (2) update each of these objects at each time step, and (3) output the plasma state periodically, both for visualization and for state preservation for restarts. The architecture of any plasma simulation code must enable these steps.

The VORPAL architecture incorporates the strict layered approach, in which no package can depend on any software layers at its level or above (see Fig. 1). This eliminates circular package dependencies. At the bottom of the hierarchy are classes that support basic mathematics and other needed objects that are independent of the physics involved. These are independent of other VORPAL libraries and could conceivably be reused outside of a plasma or fluid simulation code. On top of that is a framework layer consisting of the single package `vpbase`, in which the basic object interfaces are defined and where some basic implementations are provided. The next layer up consists of implementation classes. An example is the set of classes that implement the finite-difference-time-domain integration of the EM field on a Yee mesh. At present, VORPAL contains five basic implementation packages, those for EM fields, particles, fluids, I/O, and messaging as needed for parallelism. The next package up is the combining or control package, which contains the class definitions for the *domain objects*, which hold one or more of the various implementation objects. Finally, the top software layer consists of the package containing the executable, which creates a domain object and runs the execution loop, and two packages of utilities for post processing. One contains executable programs for data analysis and the other is a collection of scripts used to run all the executables in the top software layers and visualization programs written for IDL and OpenDX.

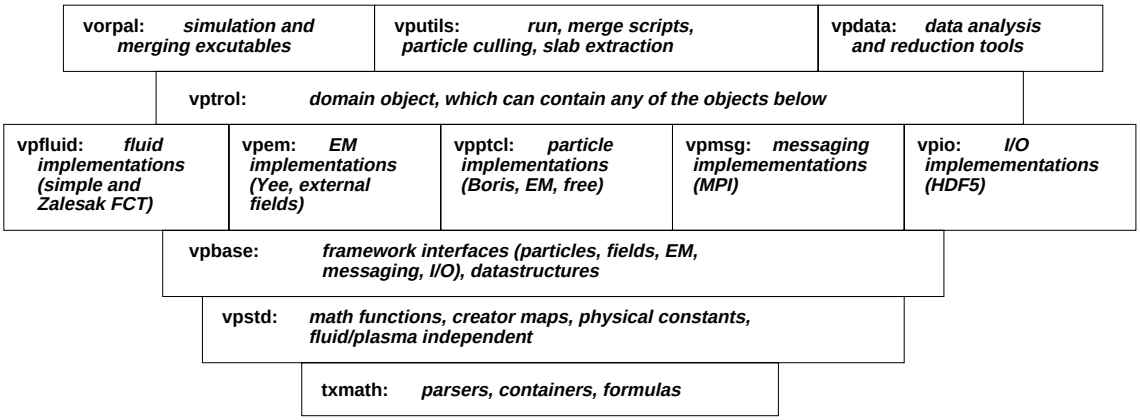

Fig. 1. Software layers in VORPAL libraries.

In this section, we discuss the simulation life cycle, then the architecture, giving a brief discussion of each of the software layers. We also provide a final subsection where we note some of the coding techniques used in VORPAL. A full listing of all VORPAL classes with on-line documentation is available at [11]. As a name-space mechanism, we prefix all VORPAL classes with "`Vp`".

2.1. Simulation life cycle

As noted in multiple places [1,3], the simulation life cycle is quite simple, consisting of initialization, dynamical evolution, and termination. In the initialization, a domain object is created. At the setting of its parameters, the domain object is told what it contains, and thus it can create each of its needed objects, such as fields, particles, boundary conditions, etc. Then the evolution loop begins. In the evolution loop, one updates the particles, charged fluids, and electromagnetic fields. Each of these updates can be multifaceted. For example, the field update in the interior integrates the equations of motion, while at the edges, it can involve the setting of a boundary condition. Periodically, a complete set of data is output, both for detailed examination and for run restoration, e.g., in case the computer fails. Finally, the evolution loop completes, the last data file is output, and the domain object is destroyed. In the process of the last step, the domain object destroys each object that it contains.

For simplicity, the only existing domain object in VORPAL assumes that the particles and charged fluids interact through only the electromagnetic field. With this assumption, one can simply update the particles one set at a time, then the charged fluids, or vice versa. After that the fields are updated. The order of update within the particle sets or fluid sets is unimportant. Provided each individual update is second order in the time step, Δt, then so is the update of the entire loop.

2.2. Coding techniques and the basic classes

As noted above, in the lowest level packages are the classes that are independent of the physics. We describe just of few of these here, as these techniques may be useful in the code development of other projects.

The Tech-X OptSolve++ library [7] provides a useful package of classes. In particular, the `TxAttributeSet` and `TxHierAttribSet` can be used to describe any of the objects within VORPAL. A `TxAttributeSet` is a collection of named integers, doubles, strings and vectors of those types. A

TxHierAttribSet derives from the TxAttributeSet and adds a vector of TxHierAttribSet's. Thus, a TxHierAttribSet has a recursive ownership chain. Any object (e.g., double, vector of doubles, etc.) within a TxHierAttribSet can be set or retrieved by name, through methods such as getParam(string). The recursive ownership chain allows a single TxHierAttribSet to define a domain object, with its owned TxHierAttribSet's describing each of the domain's owned objects, etc.

An important feature of the TxHierAttribSet objects is that they have methods for setting themselves through an XML-like string. Hence, the VORPAL input file simply contains such an XML-like string. Parsing the input file consists of reading the entire input file into one string, then giving that to a TxHierAttribSet for its initialization.

For example, the section of the input file for a cold fluid that is initially uniform may look like

```
<Fluid electronFluid>

kind = pcZalCRFluid
charge = -1.6022e-19
mass = 9.109e-31
<InitialCondition InitCond0>
lowerBounds = [ -1 -1 -1]
upperBounds = [100 100 100]
kind = constant
indices = [0]
amplitudes = [1.5e-18]
</InitialCondition>

</Fluid>
```

For use within VORPAL, we have further specifications of the format of the XML string. The opening tag for the above case means that the object to be created derives from the interface corresponding to the Fluid object of name electronFluid. This attribute set contains a string named kind with the value "pcZalCRFluid", which denotes that we are to create a VpPCZalCRFluid object, which uses the Fluid interface. Vectors of integers determine the upper and lower bounds of the region to be initialized. The two doubles named charge and mass with the values -1.6022×10^{-19} and 9.109×10^{-31} denote the properties of the fluid. The lines between the InitialCondition tags describe the initial condition of a uniform fluid density of 1.5×10^{18} m^{-3}, which is to be applied at startup.

The lowest package in the VORPAL hierarchy is the vpstd package. This package contains the MakerMap collection of classes, which allow one to construct objects from a string defining the object. It contains macros for vector operations such as cross products and dot products. It contains an object that holds the appropriate physical constants, such as μ_0 and ϵ_0 of Maxwell's equations. Finally, the vpstd package contains a set of space–time functions that can be used for initialization or for setting boundary conditions.

In the attribute sets used to handle the inputs into VORPAL, an object is described by its interface name and a string that is associated with the specific-derived class for the object. Normally a series of conditional statements would be used on the string during the actual creation of the object. This method is cumbersome, especially when one begins to add new derived classes to represent new models or algorithms. VORPAL's solution to this involves a group of classes that create objects given a string.

The first of these classes is VpMaker. VpMaker is templated over two classes, B and D where D is derived from B. The maker class has a method, getNew(), that creates an object of type D and returns a pointer of type B to that object. A second class, VpMakerMap, which is templated over the base class, contains an associative array of VpMaker's keyed by the string. It also has a getNew(std::string) that accepts a string as an argument. Upon receipt of the string, this method searches through its

associative array until it finds the correct VpMaker and then calls the getNew() for that maker. The VpMaker class is designed so when a maker is constructed, it registers itself with the associative array of the correct VpMakerMap. Thus, once a new class is created using a specific interface, all that has to be done to make it available is to create a VpMaker for the new class.

For each basic interface in VORPAL there exists an associated maker map. By calling the getNew() method, which takes the kind keyword mentioned above as an argument, the correct type of object is created with a pointer of the right interface. Using the above example of an electron fluid object, the kind key word defines what type of fluid object to create. The following code determines what type of fluid to create and then generates a pointer to an object of that type.

```
string kind = attribs.getString("kind");
VpFluid<FLOATTYPE, NDIM> *crf =
VpMakerMap<VpFluid<FLOATTYPE, NDIM> >::getNew(kind);
```

The attribs object is the hierarchical attribute set that was created from the input file. The getNew() method creates a default object whose type is determined by the kind key word and then returns a pointer of type VpFluid. The maker map is templated over the interface class so only one maker map class needs to be implemented. The different types of maker maps come from different values of the template parameter B. Since different models sharing the same interface may have very different input parameters, all the interface classes must have a default constructor. Any details, parameters, or references needed by the class are set later with other methods.

VORPAL often needs to calculate various space–time functions that depend on internal parameters, such as a function representing the electric field of a laser pulse, which has a length, wavelength, etc. We treat these abstractly as space–time functor's, objects for which the operator() method has been overloaded to take a FLOATTYPE array that represents a position and a FLOATTYPE that represents time. The base class is VpSTFunc, which defines the interface through having an abstract operator() method. This interface is then implemented in a number of derived classes for particular functions, each of which can be constructing by string using the above VpMakerMap technology. The internal parameters of the function are then set through a TxHierAttribSet.

2.3. Framework classes

VORPAL uses an object-oriented design, where the physics, numerics, and other needed elements are treated as objects. Base-class interfaces determine the information that needs to be passed between different objects in the simulation, thus determining the basic interactions among objects. Common data structures and algorithms (e.g., interpolation) are implemented in the base classes. The use of message passing and I/O objects through the base-class interfaces is implemented once in the base classes. This makes the implementation of these common objects and usages available to the derived classes, where specific physics, messaging, and I/O are implemented. This reduces maintenance and allows for the easy addition of new models and features.

The vpbase library contains the basic interfaces for the classes in the code and general abstract data objects that are common to the different models. The two principal entities in a plasma simulation are the particles and the electromagnetic field. Three interfaces are defined in vpbase for these types of objects. VpEmField provides the interface for electromagnetic fields, VpFluid is the interfaces for plasma species represented by a fluid model and VpSpecies in the interface for plasma species represented by a particle model. Other interfaces for communication and input/output are also found in this library. Data objects, such as the VpField class, which represents an object that has a known value or values at every point on the computational grid, are found here as well as other classes including the grid and support classes for the decomposition.

2.4. Implementation classes

The implementation of the various models for the electromagnetic field is found in `vpem`. There are two interfaces for electromagnetic models in `vpbase`, the basic `VpEmField` and the more specific `VpGridEmField`, which is a electromagnetic field whose values live on the grid. Since a grid electromagnetic field represents a self-consistent field, a simulation is limited to only one. However, any number of external fields can be used. The `vpem` library includes classes to represent both constant external fields and fields that depend on specific functions of space and time.

The plasma models are separated into two libraries, `vpfluid` which contains the fluid models and `vpptcl` which contains the particle models. The fluid library contains a simple flux limited version of our zero-density capable fluid model as well as a more sophisticated flux corrected version of the model. There is only one class for the particle models, but it supports a variety of different dynamics based off the policy class methods described in Section 5.2.

The `vpio` library contains classes to manage the input and output of data from VORPAL simulations. All input and output of data from VORPAL simulations is through the `VpIO` interface. At present there is only one derived class for this interface, a class that outputs the data into an HDF5 [12] file with a VORPAL-specific format. Writing to or reading from HDF5 files of a different format (e.g., conforming to emerging standards for storing simulation data) could be implemented in different classes, with no change needed for the basic framework objects or the physics implementations. Other data formats such as netCDF could be added by developing a input/output class that inherits from the `VpIO` interface.

Messaging between processors is managed by classes contained within the library `vpmsg`. Messages between processors are done through the `VpMsgCntr` interface. The message center contains classes that are responsible for sending and receiving the actual messages. Here the only implementation (in a derived class) at present is through the standard Message Passing Interface (MPI). Other communication methods, e.g., shared memory or MPI through a different API, could also go into this package. In addition, messages can be passed from a processor to itself for, e.g., the implementation of periodic boundary conditions. This is done with the self messaging classes in `vpbase`, as it is generic.

2.5. Domain classes

The `vptrol` software layer contains the `VpDomain` object, which owns virtually all other simulation objects and is responsible for their creation, deletion, and updating. Before the domain object is created, two attribute sets are created that contain all the information that tells the domain what objects it contains and the details for those object. The first attribute set contains all the command line arguments from the actual system call to VORPAL, and the second contains the information from the input file. There are some simulation parameters, such as the number of time steps to take, that may be redundant. In this case, VORPAL defaults to the value specified at the command line. The domain object has methods that allow it conduct all of the stages of the simulation life cycle discussed in Section 2.1. All of these methods such as the update method or the method that dumps the data to an output file, proceed up the ownership tree by delegation, with each object calling the update method of the objects it owns. Once the simulation is finished the domain object destroys all its owned objects.

2.6. Executable and utility packages

The three top directories in the code hierarchy are the `vorpal` directory, which contains the VORPAL executable, the `vpdata` directory, which contains utilities written in C++ for post processing data, and the `vputils` directory, which contains visualization scripts. The executable reads the input file into a `TxHierAttribSet` object, the in-memory, hierarchical representation of the system to be constructed. It

then constructs the domain object and runs it through the simulation loop. The `vpdata` directory contains executables for combining the output from multiple processors into a single file and for extracting samples of field and particle data from the combined files. The `vputils` directory contains `sh`, OpenDX, and IDL scripts to post process and visualize data generated by VORPAL.

2.7. Other features

Classes in VORPAL are templated over precision. This allows us to set the precision of the simulation at run time. Normally one would run in float precision since float precision takes half the memory as double precision and floating point arithmetic takes less time on many processors. There are situations where double precision is needed. In highly relativistic simulations the difference between the velocity and the speed of light could be smaller than float precision. On some processors, float precision arithmetic is significantly slower than double precision arithmetic.

In order to study the physics of highly relativistic systems, in particular beam physics, we have incorporated a moving window [13,14] into VORPAL. The user sets the Cartesian direction for the moving window and the distance at which the moving window will start. During the simulation, when a light pulse would have propagated from the front edge of the moving window direction to a distance set in the input file, the moving window begins, shifting the simulation region at the speed of light. At each crossing of a cell by the imagined light pulse, the grid lower bound is incremented by a cell width, fields are moved over one cell, particles now outside the grid are removed, and new quiet particles and fields are loaded in at the front edge of the simulation. This allows the user to study the propagation of a relativistic beam by following it as it propagates, rather than requiring a region that is long in the direction of propagation. As with the domain decomposition, the code for the moving window is implemented in the base classes so any new objects added to the code will inherit the moving window.

To achieve our goal of making VORPAL work on multiple UNIX platforms (including Mac OS X), we make use of the GNU project's automake and autoconf utilities. These tools allow one to create a configure script that when run, checks for the location and availability of libraries, compilers and other software that VORPAL needs to compile. The script then generates `Makefile`'s appropriate for the configuration of the build machine. We have also compiled VORPAL on Windows platforms with the Metrowerks compiler.

3. Arbitrary-dimensional coding

Templating over dimension allows one to support 1D, 2D, and 3D simulations with a single code base. The dimension of the simulation does not have to be specified until run time; this allows one to simulate a problem in 2D rapidly to get qualitative results, and then with no changes to the input file move to a 3D simulation for more detailed results.

Several challenges exist in developing an arbitrary-dimensional plasma physics code. The first is the field updates. Normally in a code where the dimension is fixed, the fields would be stored as multi-dimensional arrays and the updates would be done using nested loops. This cannot be done for a arbitrary-dimensional code as at the outset the dimension of an array and the number of nested loops is unknown. The other challenges are the interpolation of field values and the weighting of particle currents to the grid. Interpolation varies because the number of field values to interpolate differs in different dimensionalities. Two fields are required for one dimension, 4 for two dimensions, and 8 for three dimensions. Weighting the particle currents presents a problem due to the relation of the currents with the grid. The current along a simulated direction is at the midpoint of the cell edge in that direction. Weighting particle currents along a simulated direction requires the determination of the particle current crossing the cell mid-face orthogonal to that direction and amounts to nearest-grid-point weighting along the direction of the current and linear

weighting in the other directions. In contrast, a current along a non-simulated direction exists at the cell corners, and linear weighting is used in all directions.

To solve the problem of field updates, a combination of recursion and template meta-programming is used. An object is created to update the field one cell at a time. This object can be moved to any location on the grid by incrementing an index that corresponds to its location on the grid. The updater object is then walked through all points on the grid by a "walker class" that uses recursive function calls.

The updater objects who are responsible for updating the field values at a cell use a method called `updateCell()`. Typically this object represents the finite differencing of some differential equation that represents the dynamics of the object in question. To represent the components in this finite difference, we developed a generalization of an iterator [8], that allows us to deal with the problem of indexing an array in an arbitrary dimension. This iterator acts as an index to some location on the grid. The updater object contains a group of iterators for all the independent fields, a vector of weights for those iterators and another iterator for the dependent field located in the correct positions to re-create the finite difference equation. For linear finite difference equations a generic updater class exists. However, for non-linear models, such as the fluid model, updater objects specific to the model must be used.

Once one cell is updated, the updater object needs to be moved to the next cell. This is done using a `bump()` method. The analogy in one dimension is the incrementing or decrementing of the index of an array. In a multi-dimensional code the bump takes a direction and an amount to increment as arguments. This allows the updater object to be bumped to any location in the grid. The iterators also have these bump methods, so whenever an updater object is bumped to a new location, the iterators it owns are bumped in the same direction by the same amount. The iterators contain a 1D index that corresponds to its location on the grid. By knowing the strides of the grid in all dimensions, the iterator knows how much to increment the index to produce the correct index displacement corresponding to an increment in any direction.

To move the updater objects over a grid of arbitrary dimensions, we create "walker" classes that use recursion and template specialization to walk the holder objects through the grid. These classes are templated over dimension, direction, and the updater class. They have an update method which recursively calls the update method for the lower dimension. We start by defining the walker class for a general dimension and direction. This includes the recursive call to the walker class of next lowest direction.

```
template<int DIM, int DIR, class UPDATER>
class VpWalker {
   static inline void walk(VpSlab rgn, UPDATER updater) {
//  Do loop over next direction
      for(i=0; i<rgn.upperbound[DIM-DIR]; i++) {
         VpWalker<DIM, DIR-1, UPDATER>::walk(rgn, updater);
//  Bump to the next row
         updater.bump(DIM-DIR);
      }
// Bump back to the beginning
      updater.bump(DIM-DIR, -rgn.upperbound[DIM-DIR]);
   }
}
```

The walker is then specialized in the first direction to actually call the update method of the updater object. Since everything is inlined, the recursive functions are removed when sufficient compiler optimizations are called. The compiler is effectively creating the needed nested loops for us.

```
template <int DIM, 1, class UPDATER>
class VpWalker {
   static inline void walk(VpSlab rgn, UPDATER updater) {
// Do loop over next direction
      for(i=0; i<rgn.upperbound[DIM-1]; i++) {
         updater.updateCell();
// Bump to the next row
         updater.bump(DIM-1);
       }
// Bump back to the beginning
      updater.bump(DIM-1, -rgn.upperbound[DIM-1]);
   }
}
```

Although the interpolation of the fields requires a different number of fields with different weights for different dimensions, it follows a pattern that lends itself to recursion. For example, the interpolation of the electric field to the location of a particle in two dimensions is given by

$$\mathbf{E} = (1-w_i)(1-w_j)E_{i,j} + w_i(1-w_j)E_{i+1,j} + (1-w_i)w_jE_{i,j+1} + w_iw_jE_{i+1,j+1}, \tag{1}$$

where w_i and w_j are the distance from the point i,j to the particle. Eq. (1) can be rewritten as follows:

$$\mathbf{E} = (1-w_j)\left[(1-w_i)E_{i,j} + w_iE_{i+1,j}\right] + w_j\left[(1-w_i)E_{i,j+1} + w_iE_{i+1,j+1}\right]. \tag{2}$$

This is just an interpolation in the ith direction followed by an interpolation in the jth. This result generalizes to three dimensions allowing the interpolation to be done with recursion and meta-template programming in a similar manner as the field updates.

For the current explicit electromagnetic implementations in VORPAL it is assumed that system itself is 3D but the physical quantities do not vary in the remaining directions. This means that regardless of dimension, vector quantities such as the currents and the electromagnetic fields always have three components. This complicates efforts to develop an arbitrary-dimensional method to weight the currents. We use the current weighting scheme of Villaseñor and Buneman [15] to calculate the currents in the directions that lie along simulation dimensions. Since this method depends on determining how much charge enters or leaves a cell, it cannot be applied to currents in the symmetry directions. In those cases, the currents are determined by the product of the charge density at the grid point times the velocity at that point. So the weighting for the current in the z-direction is quite different between a 2D and 3D simulation. To deal with the need for two different weighting schemes, we use `switch` and `case` statements to deal with each dimension individually. It may be possible to deal with this problem in an arbitrary-dimensional matter, however it was expedient not to pursue this at the time. There are only three cases to deal with, and the compiler removes the `switch` and `case` with sufficient optimization. We may return to this problem at a later date to try and find a more elegant solution.

The advantage of the VORPAL arbitrary-dimensional methodology is that one has a single code base for all dimensionalities, with little dimension-dependent code. This reduces maintenance requirements. The use of recursion usually implies a loss in performance due to excessive function calls and repeated switch statements can also adversely affect code speed. By examining the object files produced by the compiler, we have confirmed that the use of template specialization and sufficient compiler optimizations will inline all the recursive function calls and switch statements used in our arbitrary-dimensional coding methodology.

4. Parallelization

VORPAL is designed to run as both a serial code for single-processor workstations and as a parallel code for systems that support MPI. As part of our general philosophy of flexibility, we have developed a general 3D domain decomposition. This general domain decomposition allows for static load balancing and makes it possible to incorporate dynamic load balancing in the future. To minimize the overhead that occurs with messaging, VORPAL overlaps computation and communication as much as possible. The abstract interface for message passing in VORPAL is defined low in the class hierarchy, and message passing is done low in the hierarchy through this interface. This means when new classes are added, they will automatically inherit the message passing from the appropriate base class.

Load balancing is an important issue for PIC codes since the particles are not necessarily distributed in a spatially uniform manner. For example, in a simulation of a particle beam propagating through an accelerating structure, the particles are concentrated in the vicinity of the beam. Also the computer being used may have processors of differing capability. This is especially true of Beowulf clusters, as they become upgraded over time. If new processors are added to a Beowulf cluster over time, Moore's law tells us that there will be a significant imbalance in the computational power between the old and new processors.

Most mesh-based codes including VORPAL use a data parallel decomposition scheme to parallelize the simulation domain [16]. This means each domain is assigned a certain number of grid points and particles for which it is responsible for updating. A typical decomposition of a mesh-based code is done by separating the domains by cutting through the simulation region using planes. A typical 2D domain decomposition of a 3D cubic region is shown in Fig. 2. The particles in VORPAL are distributed among the processors by assigning a particle to the domain that owns the grid cell in which it lives. The total computational load for the domain is then load balanced. This is referred to as unitary load balancing [17].

A more general decomposition than the one in Fig. 2 is needed for load balancing. To balance the computational load from the four domains, the time the first domain needs must be matched to the time the remaining three domains need. This implies three conditions for equality of the computing time used by each domain. However, there are only two movable planes, thus insufficient freedom for satisfying the three conditions.

By using a more general decomposition where the planes separating the domain do not have to extend completely across the simulation region, load balancing is now possible. In Fig. 3, the decomposition planes perpendicular to one direction cut the entire region, but in the second direction they break at each of the planes of the first direction. Having three movable planes to satisfy three conditions makes load balancing

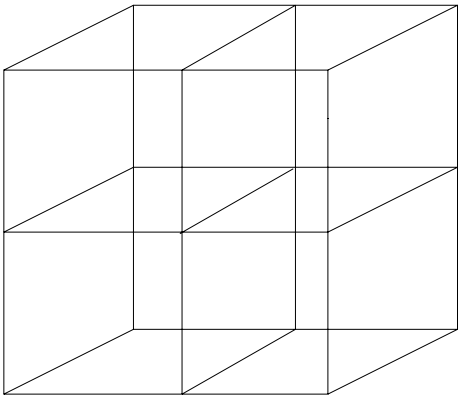

Fig. 2. Standard 2D decomposition of a 3D cubic region.

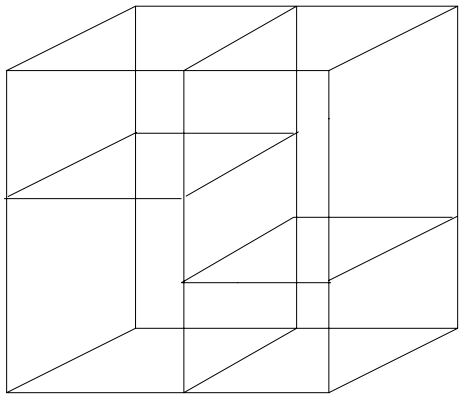

Fig. 3. Fully general 2D decomposition of a 3D cubic region.

possible. One method of achieving the decomposition shown in Fig. 3 is to bisect the simulation region and the bisect the sub-regions in an iterative manner until the the number of domains equals the number of processors [18]. This method has been used to load balance particle based codes [19]. The hierarchical domain decomposition [17] also allows for such general domain decompositions by choosing a direction, decomposing that direction into a number of domains, then decomposing those domains along a perpendicular direction and so on. A good review of a large variety of methods for domain decomposition and load balancing of PIC codes can be found Carmona and Chandler [20].

VORPAL's domain decomposition is even more general, allowing any domain decomposition that consists of a collection of slabs. This is done using set theoretical ideas. We have defined the concept of a `VpSlab`, which is a logically cubical (bounded by six planes – orthogonal parallelepiped). Each domain is described by two `VpSlab`'s, its physical region, `physRgn`, plus a region that includes a layer of surrounding guard cells, `xtndRgn`. The `physRgn` is the region over which the domain must solve the dynamics. To do this it must know the field at beginning of the time step in its `xtndRgn`. Thus the required communication is that each domain receive the values in its `xtndRgn` from the processors that have calculated those cells. Similarly, a processor must send all the cells in its `physRgn` that belong to another

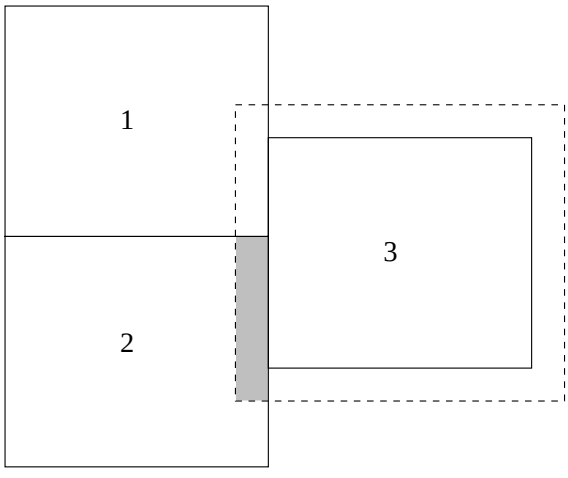

Fig. 4. The intersection of an extended region with a neighboring domain's physical region.

processors xtndRgn. We can see that all that needs to be done is to figure out the intersection of the domain's xtndRgn with a neighboring domain's physRgn. This intersection is itself a VpSlab, and it contains the cells which the neighboring domain needs to pass to the domain in question. In Fig. 4, the solid lined rectangles are the physRgn's of the various domains and the dashed line rectangle is the xtndRgn of domain 3. The shaded rectangle is the intersection of the xtndRgn of domain 3 with the physRgn of domain 2. This is the region that domain 2 must pass to domain 3. This method of determining the send and receive regions allows any domain decomposition that is a sum of slabs. For example, consider simulating a waveguide connected to a large cavity. The cavity could be broken up in a regular manner while the waveguide is broken up into a line of slabs in one direction.

Some algorithms require a third VpSlab to be associated with the domain, referred to as the xtndPlusRgn. This is the same as the xtndRgn except the guard cells need to be two cells thick on the upper side of the region in all directions. The Zalesak FCT fluid model requires these extra cells to get the slope of the density for the flux correction. The particles need the current field defined on these extra cells since a particle that enters an outer cell of the xtndRgn can deposit current to a cell interface in the xtndPlusRgn. The extra cells are needed only on the upper side since the current associated with a cell is the current located on the lower cell face.

Load balancing the standard PIC algorithm has in the past required the computational load from the field solve and the particle push be balanced separately [21]. The reason for this is that all the currents from the particles must be known before the electromagnetic field is updated and the electromagnetic field must be known at all points before the particle push is done. If the domains are arranging in a irregular manner such that the particle load is balanced, the field updates are now unbalanced. One method of dealing with this issue is the dual-domain decomposition method [22]. In this method, the particles and field have separate domain decompositions. The computational loads for each update can then be balanced separately. This method has the complication that each processor must store the field values it is responsible for updating, plus any field values that are required for the particle updates. If the particle domains are considerably different from the field domains, there will be a large redundancy in data storage and excess communication for the field values that overlap the particle and field domains.

Since VORPAL's particle and field domains coincide, the only data redundancy and communication that occur are in the layer of guard cells around each domain. However the PIC algorithm must be modified so total computational load for each domain can be load balanced rather than balancing the particles and fields separately. We use an algorithm similar to one used by the ICEPIC code [21,23]. This involves overlapping the communication and computation between domains at at two different levels. The first is an internal overlapping in the fluids and fields. By using the VpSlab class one can specify any subregion of the physRgn. We split the physRgn into an edge region, which is either one or two cells thick depending on the algorithm being used, and a center region, which consists of all the remaining cells. During an update the edge region is updated first, as the data of the cells needs to be passed to neighboring domains. After these cells have been updated, all the sends go out to the neighboring processors. Now the center region is updated while communication is proceeding, with the message receives done after the all the center cells have been updated.

By staggering the updates and communication calls of the particles, fluids, and fields in the right manner, further overlap can be achieved. At each time step, the particles are updated first, and their associated currents are deposited to the sumRhoJ field, which collects the total charge and current from all the particle species and charged fluids. The particles that have moved to the domains of other processors are appropriately sent. The current contributions to cells in the xtndRgn are sent to the appropriate processors. Now the fluid updates are performed, with the communication overlap internal as described in the previous paragraph. The fluid update includes depositing the fluid charge and current to the grid. For the Yee electromagnetic field, the first half-step update of the magnetic field can be performed, since it does not depend on the currents. Then the currents are received and the electric field can be updated. Both the

electric and magnetic updates are overlapped internally as discussed in the previous paragraph. Finally after all the physics objects have finished their updates, the particles are received. Thus, computation overlaps communication, and many different types of communication (for field solves, currents, and particles) are occurring simultaneously. Further overlap might be achieved by receiving and updating sent particles after the particles currently existing on the processor are updated, but we have not so far found this necessary. A time line of a VORPAL update showing the order of communication and computation is shown in Fig. 5.

We ran scaling tests (increasing the number of processors for constant problem size) of VORPAL on the IBM SP at the NERSC supercomputing center to test its performance in a parallel environment. There are a variety of reasons that parallel scaling is lost. Amdahl's law states that parallelism breaks down when the serial work becomes comparable to the parallel work divided by the number of processors. Another

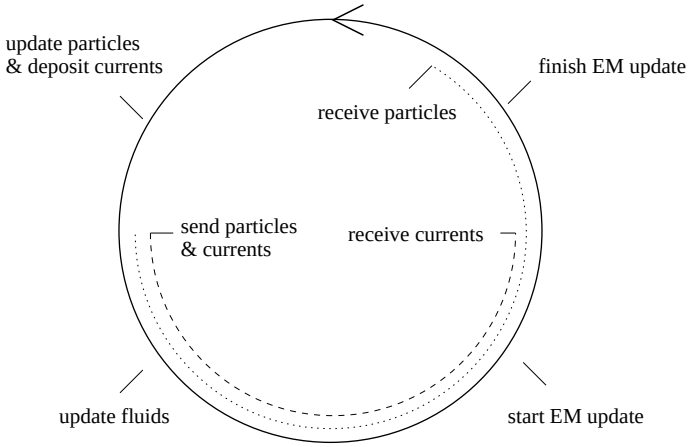

Fig. 5. Time line for VORPAL update showing communication and computation overlap.

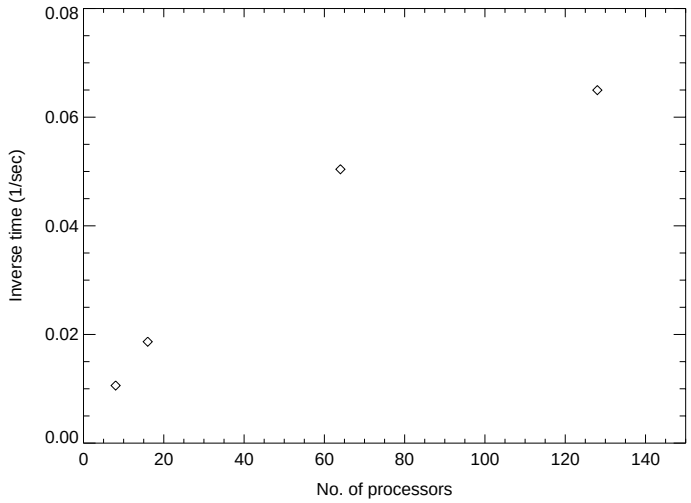

Fig. 6. The inverse time plotted against the number of processors for the $200 \times 100 \times 100$ cell electromagnetic only scaling run.

common limiting factor in data decomposed parallel codes is that as the ratio of the number of surface cells of a domain to number of volume cells gets small, there is less computation to overlap with the communication.

The first scaling test was an electromagnetic only simulation of $200 \times 100 \times 100$ cells. In Fig. 6, we show the scaling as the number of processors varies from 16 to 128. The scaling begins to break down at 64 processors. For larger problems, the scaling extends to a greater number of processors. Fig. 7 shows the scaling for a VORPAL electromagnetic simulation of $400 \times 200 \times 200$ cells. For this larger problem good scaling to 256 processors is observed.

VORPAL scales even better once particles are added. Fig. 8 shows scaling tests for a $200 \times 100 \times 100$ cell PIC simulation with 5 particles per cell. The scaling is good to around 512 processors. This better scaling is

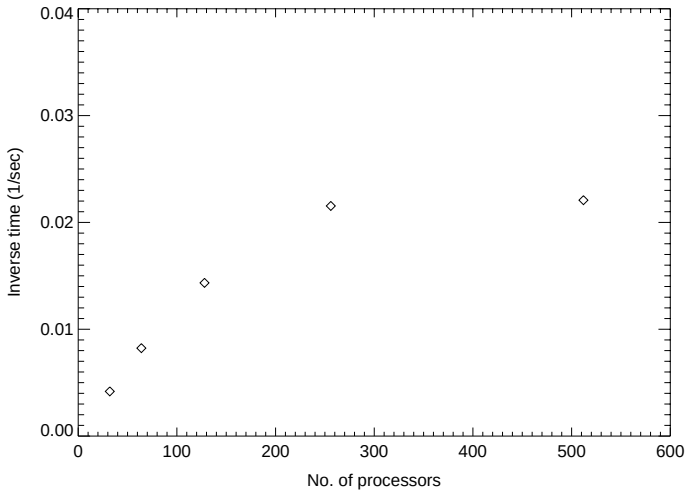

Fig. 7. The inverse time plotted against the number of processors for the $400 \times 200 \times 200$ cell electromagnetic only scaling run.

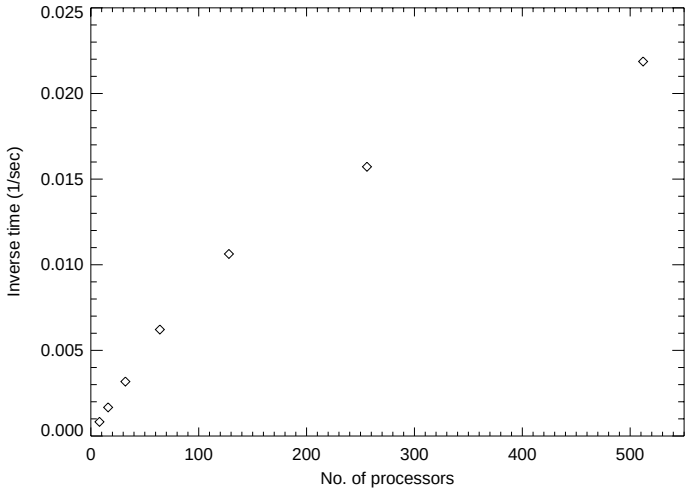

Fig. 8. The inverse time plotted against the number of processors for the $200 \times 100 \times 100$ cell PIC scaling run.

expected as there is more parallel versus serial work when running PIC, and there is more computation relative to communication; PIC simulations require less communication than pure field simulations, as not every edge particle crosses the boundary at every time step, but every edge field component must be communicated to neighboring domains at every time step. We have found for larger problem sizes, PIC simulations have good scaling past 1000 processors.

5. Available implementations

Multiple models for both the electromagnetics and the plasma exist in VORPAL. Aside from some basic constraints, such as having only one grid electromagnetic field, both the plasma and the fields can be represented by multiple models in the same simulation. As an example, a hybrid simulation of optical beam injection in LWFA could represent the injected particles with a PIC model and the bulk plasma with a fluid model. There are other possibilities, such as modeling certain species in the plasma with a fluid model and others with a PIC model. The same is true for the electromagnetics. To study the behavior of a plasma inside a solenoid, VORPAL can model the solenoid's field as constant external field but it also solves for the self consistent field generated by the plasma.

5.1. Electromagnetic field implementations

As we mentioned in Section 2.3, the electromagnetic models all derive from a base class `VpEmField`, which defines the basic interface for a electromagnetic field. The basic interface has methods for constructing and updating the object. It also has methods for obtaining the values of the electric and magnetic fields at a set of points. A more specific electromagnetic base class, `VpGridEmField`, assumes that the fields live on specific points on the grid. It has all the behaviors of the basic model, using interpolation to find the field values at any point and requiring knowledge of the charge and current density in order to update itself.

Currently there is only one electromagnetic model in VORPAL that inherits from `VpGridEmField`. This model uses a finite difference time-domain solver based on Maxwell's equations using a Yee mesh [24] to provide second-order accuracy. Faraday's equation,

$$\frac{\partial \mathbf{B}}{\partial t} = -\nabla \times \mathbf{E} \tag{3}$$

and the Ampere–Maxwell equation,

$$\frac{\partial \mathbf{E}}{\partial t} = c^2 \nabla \times \mathbf{B} - \frac{\mathbf{j}}{\epsilon_0} \tag{4}$$

are updated through the finite-difference scheme. The remaining Maxwell's equations,

$$\nabla \cdot \mathbf{E} = \frac{\rho}{\epsilon_0}, \tag{5}$$

$$\nabla \cdot \mathbf{B} = 0 \tag{6}$$

need not be solved provided the charge and current densities obey the finite differenced continuity equations [15], which we assume to be the case.

In the Yee scheme [24] the second-order accurate fields are known at different points within the cell. The second-order accurate electric field in direction i is known at the midpoint of the cell edge along direction i. The second-order accurate magnetic field in direction i is known at the center of the cell face orthogonal to direction i. The second-order accurate charge is at the cell corner, while the second-order accurate currents

Table 1
The position of the electromagnetic field components relative to the lower corner of the cell for a 3D Yee mesh in VORPAL

Component	Cell location
E_0	(1/2, 0, 0)
E_1	(0, 1/2, 0)
E_2	(0, 0, 1/2)
B_0	(0, 1/2, 1/2)
B_1	(1/2, 0, 1/2)
B_2	(1/2, 1/2, 0)

are at the same locations as the electric fields. The specific locations for the components of the electric and magnetic fields in three dimensions are given in Table 1. This same table gives the cell locations for lower dimensionality by simply ignoring the unused dimensions. (All three components of the electric and magnetic fields are kept when simulating 1D or 2D systems.)

To simplify the interpolation for obtaining the field at an arbitrary point in the cell, an intermediate set of *nodal fields* are obtained by appropriate averaging of the values on the Yee mesh to obtain values at the cell corners. For example, the nodal value of E_i is just the Yee value if i exceeds the dimensionality. Otherwise it is the average of the Yee E_i values of the cell and the cell below. Nodal fields at domain edges are calculated using Yee field values stored in ghost cells at the edge. These ghost cell values are set according to the relevant boundary conditions or by inter-processor communication.

Other non-self consistent electromagnetic models exist for VORPAL to model various types of external fields. Fields that are constant in space and time and ones that have a specific functional depend on space and time are available. A electromagnetic model exists that allows one to combine two or more models, typically a group of externally applied models and a self consistent model. This allows us to pass a single electromagnetic field to the plasma models.

5.2. Particle implementations

Since fluid models deal with only the bulk velocity of the plasma, they do not incorporate kinetic effects coming from the distribution of velocities among the particles in a plasma. However, full N-body simulations are not practical, since one cannot simulate enough particles to represent the sizes and densities of the plasmas we are interested in studying. A common solution to this problem is making use of macro-particles. Simply put, rather than track the motion of each particle individually, we put the particles into groups and track the motion of the entire group. This idea can be further improved by having the particle group density spread out over a region of space centered about its center of mass, rather than having all the density at a point. The change in the particle momentum is then calculated by considering the impulse from any relevant forces and the particle position is updated by advanced the particle according to its current velocity. These models are referred to as Particle-In-Cell (PIC) models [1,2].

For grid EM fields in VORPAL, this requires the field values be interpolated to the location of the particle group. For the PIC model to be self-consistent, the amount of particle density that crosses each cell boundary is determined, and the corresponding current is then deposited on the grid. We use the weighting scheme of Villase nor and Buneman [15] to find these currents. The electromagnetic field is then updated using these current values.

To ensure computational efficiency in the particle updates, we update particles in groups, and we avoid virtual functions through use of policy classes. Particle positions and velocities need to updated at each time step. The details of how these quantities are updated will vary depending on the model. In the traditional model of inheritance, there would be methods, `accel` and `move`, for each of these updates, and these methods would have to be virtual so that different models could overload them to generate different dy-

namics. Thus, there could be a cost from the function call, and from the fact that the function is virtual. We amortize these costs by storing the particles in groups, each of which contains vectors of position vectors and velocity vectors, and updating those groups. (This has advantages over a single large array when particles are created and destroyed throughout the simulation.) Still, there remain several calls to the `accel` and `move` methods at every time step.

We further eliminate these computational costs by using policy classes to implement a variety of different particle dynamics without the disadvantages of using traditional inheritance. Basically the methods responsible for accelerating and moving the particles are not implemented in the particle class. They are implemented in a policy class which the particle class is then templated over. In the following pseudo-code, we see that the acceleration and move are delegated to the methods of the `MOVER` class, which is the policy class that determines how the particle velocity and position are updated:

```
template <class MOVER>
class VpDynSpecies {
   void update(double t){
      (loop over all particle groups){
         ...
         MOVER::accel(ptclGrp);
         ...
         MOVER::move(ptclGrp);
         ...
      }
   }
}
```

The update method of `VpDynSpecies` loops over all of its particle groups and calls the `accel` and `move` methods. Since these methods are not virtual they are inlined away by compiler optimizations. This would allow these methods to be used even on individual particles without loss of performance due to using a function call.

A policy which accelerates the particles according to an electrostatic model would look something like the following:

```
class VpNonRelES {
   static void accel(VpPtclGroup ptclGrp){
         // v = v + qEΔt
   }
}
```

Therefore `VpDynSpecies<VpNonRelES>` is a particle class whose particles are updated using an electrostatic push. Since the policy includes only two methods, one to move the particle and one to accelerate it, adding a new policy class and, hence, a new PIC model is straight forward.

VORPAL currently has several different policy classes to update the particles. A policy, `VpFreeRel`, for free streaming particles exists that only updates the particle position, leaving the particle velocity unchanged. A non-relativistic electrostatic policy class, `VpNonRelES`, accelerates the particles using only the electric field. Both relativistic and non-relativistic policies, `VpNonRelBoris` and `VpRelBoris`, exist for the Boris push update, where both the acceleration from the electric field and the velocity rotation from the magnetic field are determined.

The number of initial macro-particles can be set in the input. One can reduce the numerical noise of the simulation by raising the number of macro-particles used. Of course this increases the computational load of the simulation. Another way of reducing the noise is in the initial distribution of the macro-particles.

Rather than placing the particles on the grid randomly, one can used a so called quiet start distribution that reduces the noise from the simulation.

5.3. Fluid implementations

The details of VORPAL's fluid model were driven by the needs of the target simulation of LWFA. In high intensity laser–plasma interactions the laser comes in from a vacuum region, and the ponderomotive force can be strong enough to blow out the plasma from regions where it initially existed. Therefore a new fluid algorithm is needed that can simulate regions of zero density. In many other applications of computational fluid mechanics, the effects of the pressure are important at some level and must be included in the model [25]. For the laser–plasma interactions that occur in LWFA, thermal effects are not important. This means we are free to use the cold fluid model for the plasma where the pressure term is neglected. Not only does this simplify the model, but since there is no pressure term the momentum equation can be rewritten so that the density does not appear anywhere in the equation. This provides a basis for a fluid model can deal with regions of zero density.

The equations of motion that are most commonly used to describe fluid behavior are the continuity equation

$$\frac{\partial n}{\partial t} + \nabla \cdot n\mathbf{v} = 0 \tag{7}$$

and the momentum density equation

$$\frac{\partial \mathbf{p}}{\partial t} + \nabla \cdot (\mathbf{p}\mathbf{v}) = qn\left(\mathbf{E} + \frac{\mathbf{v}}{c} \times \mathbf{B}\right), \tag{8}$$

where the momentum density $\mathbf{p}$ and the fluid velocity $\mathbf{v}$ are related by

$$\mathbf{p} = \gamma mn\mathbf{v} \tag{9}$$

for relativistic fluids. These equations are used because they appear in flux conservative form. Solving them numerically simply involves determining the flux that is crossing each cell interface and then updating the value in that cell accordingly. However, finding the fluid velocity requires dividing the momentum density by the fluid density. This requires the fluid density to be non-zero at all points in the simulation.

Since the cold fluid equations do not involve a pressure term, Eq. (8) can be rewritten by expanding the second term, using Eq. (7) to remove terms involving derivatives of the density, and then divide by the density to arrive at

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{u} = \frac{q}{m}\left(\mathbf{E} + \frac{\mathbf{v}}{c} \times \mathbf{B}\right), \tag{10}$$

where $\mathbf{u} = \gamma\mathbf{v}$ is the relativistic velocity.

Eq. (10) without the Lorentz force term simply describes how the velocity field advects or is moved along by the fluid flow. By using operator splitting we can separate the velocity update into the advection of the velocity by the velocity field and the acceleration by the electromagnetic field. The electromagnetic acceleration is handled by using the standard Boris push. To determine the momentum advection we simply calculate the distance, $\mathrm{d}x$, that a cell interface moves during one time step and then shift the velocity values by a weighted interpolation with $\mathrm{d}x$,

$$u_i \to \frac{\Delta x + \mathrm{d}x}{\Delta x} u_i - \frac{\mathrm{d}x}{\Delta x} u_{i-1}, \tag{11}$$

where Δx is the grid spacing. To achieve second-order accuracy in space, dx is calculated by interpolating the velocity to the distance dx. After some simple algebra, we arrive at the expression,

$$dx = -v_i(1 - v' dt)dt, \tag{12}$$

where v_i is the velocity moving into the ith cell and v' is the slope of the velocity between the two cells.

Since Eq. (7) is in flux conservative form, flux transport can be used to do the density update. A naive approach to determining the particle flux at the cell boundaries would be to again determine the distance, dx, that the cell interface moves in one time step. This is done in a similar fashion to the velocity advection, but with the v_i being replaced with the average velocity between the cells $\bar{v}$. The expression for dx is now,

$$dx = -\bar{v}(1 - v' dt)dt, \tag{13}$$

where $\bar{v}$ is

$$\bar{v} = \begin{cases} \frac{v_i + v_{i+1}}{2} & \text{if } v_i < 0, \\ \frac{v_{i-1} + v_i}{2} & \text{if } v_i > 0. \end{cases} \tag{14}$$

This along with the cross-sectional area of the cell determines the volume that moves from one cell to the next. By finding the average density in this volume we now have the density that moves from one cell to the next

$$\Delta n_{i+1/2} = \left(\bar{n} + n' \frac{dx}{2}\right) dx, \tag{15}$$

where $\Delta n_{i+1/2}$ is the density flowing from the ith cell to the $i + 1$ cell. $\bar{n}$ and n' are defined in the same way as $\bar{v}$ and v'. The corresponding flux between cells is given by dividing $\Delta n_{i+1/2}$ by the grid spacing perpendicular to the cell face. The flux is used to determine the current crossing the cell interface.

This naive approach gives a charge conserving algorithm. However, if the density difference between the two cells is large enough, then the density in the one cell can achieve a non-physical negative value. To combat this problem in our original algorithm, we limit the density leaving a cell to a certain fraction of the original density in the cell. By setting that fraction to a half, we guarantee the density in the cell will never go negative.

Despite the crudeness of the flux limitation for this algorithm, we find it gives reasonable results. There are other problems with this approach. Our algorithm does preserve positivity, but it does not preserve monotonicity. This means that although we prevent the density from going negative we do not prevent the formation of new non-physical extrema that result from numerical anti-diffusion. In order to preserve both positivity and monotonicity we further improve the algorithm by using Zalesak's implementation [26] of Boris and Book's flux corrected transport (FCT) [27]. The basic idea is to first calculate the flux with a first-order algorithm that is known to be positive definite and monotonic. Then higher-order corrections are determined. Then the higher anti-diffusion of the higher-order flux is limited to just prevent the generation of new extrema.

To implement this algorithm, we separate the first-order part of our original algorithm from the higher-order corrections. The separation occurs by considering the first- and second-order parts of the dx separately and rewriting the average density $\bar{n}$ as the density of the "source" cell plus a correction,

$$\bar{n} = n_i + \frac{\Delta x}{2} n'. \tag{16}$$

We can now apply Zalesak's FCT algorithm.

6. Code validation

VORPAL has been rigorously tested to ensure it produces accurate, meaningful results. A series of regression tests are included with VORPAL to validate the code and ensure consistency when the code is updated. The code has reproduced a variety of known results, including the paraxial approximation of a laser pulse propagating in vacuum and the generation of a plasma wake field by a short, high intensity laser pulse. Here we show one regression test, the validation that VORPAL correctly produces longitudinal plasma oscillations for a Gaussian potential,

$$\phi = \phi_0 \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right). \tag{17}$$

Poisson's equation gives the corresponding electron charge density for this field. The electric field is then initialized to the field described by this potential and the associated plasma density is initialized to a constant density plus the density that corresponds to the potential. The corresponding physical situation is a neutral plasma, where the motion of the positive charge carriers are neglected, and the electron density has a small perturbation that generates the electrostatic potential given by Eq. (17). The perturbation will generate a small amplitude plasma oscillation whose frequency can be compared with linear plasma theory. Neglecting magnetic and thermal effects, the angular plasma frequency for a small amplitude oscillation in an electron plasma is given by

$$\omega_{\mathrm{p}} = \left(\frac{n_0 e^2}{\epsilon_0 m_e}\right)^{1/2}, \tag{18}$$

where n_0 is the density of the plasma.

To produce a perturbation in the particles that corresponds to the Gaussian potential in Eq. (17), a small displacement is added to the positions of the particles when they are loaded into the simulation,

$$\vec{x}' = \vec{x} + \vec{\xi}(\vec{x}), \tag{19}$$

where $\vec{x}'$ is the new position to load particle and $\vec{x}$ is the position from a loading method that generates a uniform density. It is easy to show that if $\vec{\xi}$ is the gradient of some potential function, then the charge density produced by this particle distribution satisfies Poisson's equation for this potential.

An unperturbed plasma density of 1×10^{-15} m^{-3} gives a angular plasma frequency, ω_{p} of 1.784×10^9 rad/s and a plasma wavelength, λ_{p} of approximately 1 m. A square region 10 m on a side with 100 grid points per side gives a space that is 10 plasma wavelengths long with 10 grid points per plasma wavelength. The plasma is represented with macro-particles using 30 particles per cell. To comply with the Courant condition the time step is chosen to be 2.25×10^{-10} s, and the simulation is run for 75 steps corresponding to approximately 5 plasma oscillations. The electric field and particle positions are initialized to correspond to a potential with $\phi_0 = 0.001$ V and $\sigma = 1$ m. This corresponds to a peak density perturbation of about 0.2% of the bulk plasma density.

In Fig. 9 we plot the value of the electric field at one of its spatial extrema as a function of time. The data is then fitted to a simple cosine to determine the frequency of the oscillation. The fitted plasma frequency is found to be 1.791×10^9 rad/s which compares well with the predicted frequency of 1.784×10^9 rad/s. A similar simulation was done using the simple fluid model that is included with VORPAL. The same potential was used but the fluid density is initialized directly. Using the same parameters a plasma frequency of 1.792×10^9 rad/s was found, again in good agreement with the predicted results.

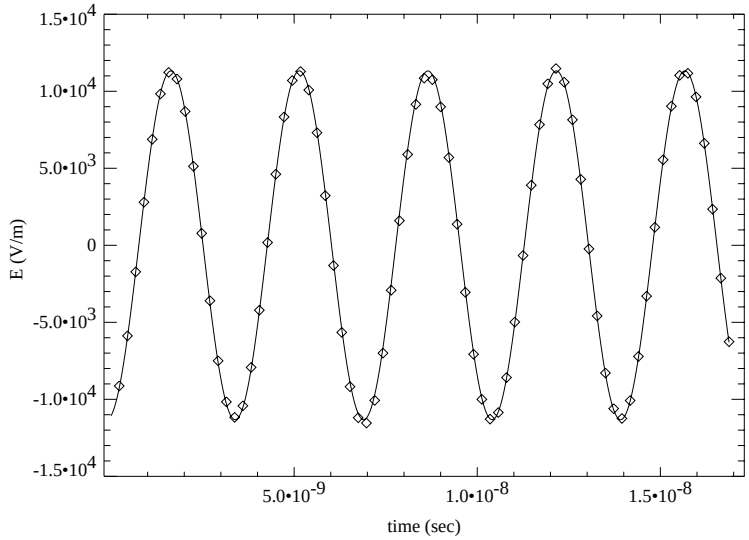

Fig. 9. The electric field versus time for the PIC validation simulation. The diamonds are the data and the line is the fit.

7. Hybrid simulation of laser wake field acceleration

The LWFA is an example of a situation where a hybrid PIC/fluid simulation would be useful, since the beam and the plasma oscillations that generate the accelerating wake field are in some sense separate entities (see Fig. 10). The beam itself is best modeled by a collection of macro-particles, but by modeling the wake field with a fluid, one can avoid the noise that is associated with modeling the bulk plasma with PIC and reduce the computational requirements of the simulation.

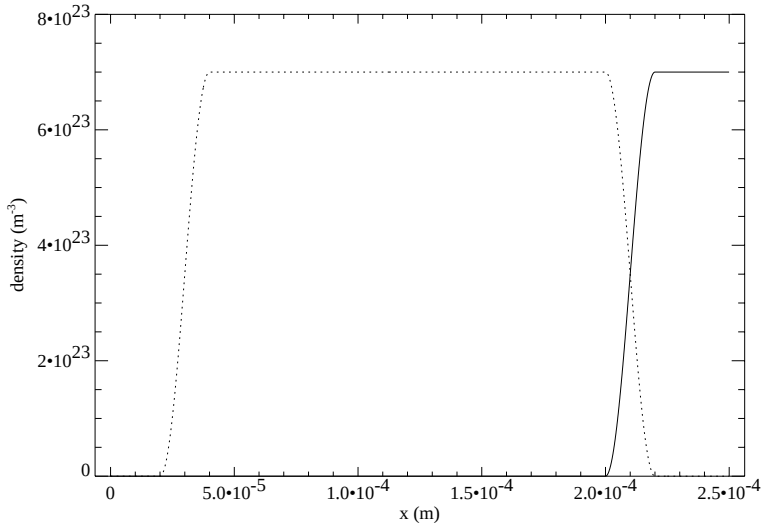

Fig. 10. The initial particle distribution and fluid density used for a hybrid LWFA simulation. The dashed line is the density profile of the particles, and the solid line is the density distribution for the fluid.

In the beat-wave (or colliding-pulse) injection scheme [28] for Laser Wake Field Acceleration (LWFA), three laser pulses are fired into the plasma. The first pulse, referred to as the pump pulse, is responsible for generating the wake field. The remaining two pulses are used to kick particles up to an energy that puts them traveling in phase with the wake field for subsequent acceleration. This injection works by firing two counter propagating pulses with polarization perpendicular to the pump pulse. One pulse trails the pump pulse at a distance that puts it in the accelerating region of the wake field. The second injection pulse is launched from the other side of the plasma. Since injection pulses have a polarization perpendicular to the pump pulse, the second injection pulse passes through the pump pulse with minimal nonlinear interaction due to the plasma. When it reaches the other injection pulse the two pulses beat. This generates a short lived large electric field and a beat potential that inject the beam particles into the wake field.

To perform a hybrid simulation of this situation we model the front of the plasma with a collection of particles long enough that injection occurs with this region. Past this region the plasma is then modeled by a fluid. When injection occurs the particles that represent the beam are traveling near the speed of light so they will remain in the simulation after the moving window is active. The remaining particles will leave the simulation as the window shifts the plasma, moving the wake field into the region where it will be modeled by the fluid. Although initially the computational cost of such a hybrid simulation is greater than a simple PIC simulation, once the moving window moves the bulk of the particles out of the simulation, only the beam particles are being updated. So the hybrid simulation reduces computational cost for long runs and reduces the noise in the simulation since the bulk of the plasma is being modeled by a fluid.

A simple hybrid simulation shows that VORPAL's ability to use multiple models to represent the plasma gives it the capacity to perform these hybrid simulations. To arrive at a plasma wavelength (c/f_p) of 40 μm we set the plasma density to be 6.97×10^{23} m^{-3}. The pump pulse and the right traveling colliding pulse are launched into vacuum from the left boundary in what we refer to here after as the x-direction, and the right traveling pulse is created adiabatically within the plasma. All the pulses have a half cosine profile in the direction of propagation and are Gaussian in the transverse direction so the electric field has the following functional form,

$$E_i = \frac{m_e c \omega a_i}{e} \cos(\pi(x - x_i - v_{gi}t)/2L_i) \exp(y^2/2w^2) \cos(k_i x - \omega_i t) \tag{20}$$

for $|x - x_i - v_{gi}t| < L_i$ where the subscript i is either p for the pump pulse (with the − sign in the argument of the last cosine), which creates the accelerating wake field, f for the forward pulse, or b for the backward pulse (with the + sign in the argument of the last cosine). The length of the pulse is L_i, and the rms width of the pulse is w_i.

We take the rms length of the pump pulse, L_p to be half the plasma wavelength and the lengths of the two colliding pulses to be half a plasma wavelength. The rms width of the pump pulse is 21.2 μm, and the width of the colliding pulse are again half as much. The wavelength of the pump pulse and the backward moving colliding pulse is 8 μm and the wave length of the forward moving colliding pulse is 8.3 μm so the two colliding pulses will beat. The backward moving pulse is created adiabatically centered at 155 μm so the plasma can respond correctly the laser fields, and the forward moving pulse trails the pump by 55 μm measured center to center. This ensures that injection will occur at an accelerating and focusing region in the wake field. The plasma starts at zero and rises as a half cosine to the bulk density over 80 μm. The plasma is represented by particles for the next 100 μm, so the injection pulses collide in a region represented by particles. Following this region is a transition region of 20 μm where the particle density drops as the fluid density rises. After we have a constant fluid density.

In Fig. 11, we see the x-component of the relativistic velocity (γv) of the particles plotted against their x positions. A beam has clearly been formed and accelerated to an energy of approximately 14 MeV in less than half a millimeter. At this point in the simulation the initial particle region has been moved out of the simulation by the moving window, and only the particles traveling near the speed of light are present. In

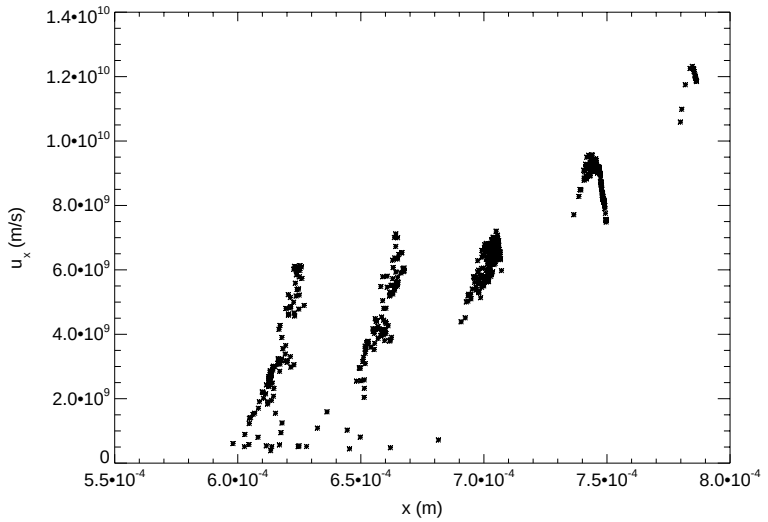

Fig. 11. The relativistic velocity in the x-direction of the particles for the hybrid colliding pulse simulation plotted in the x-direction.

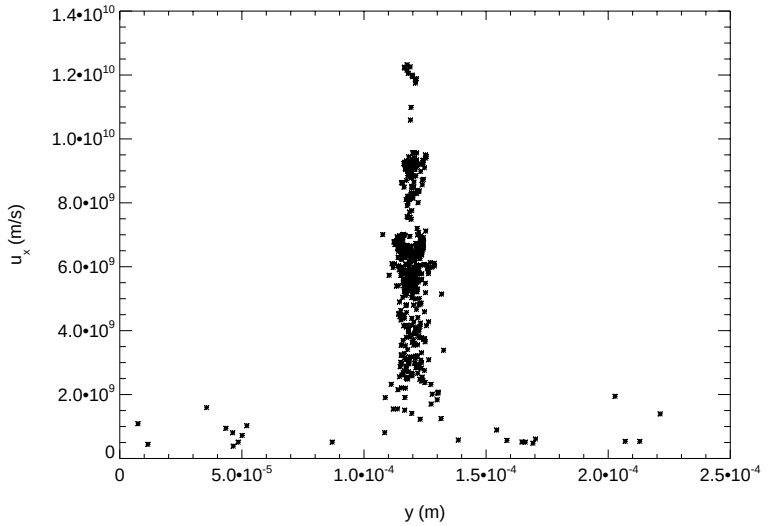

Fig. 12. The relativistic velocity in the x-direction of the particles for the hybrid colliding pulse simulation plotted in the y-direction.

Fig. 12 we see the x-component of the relativistic velocity plotted against their y positions. Here we see the beam is localized in both transverse and longitudinal directions. We also note that in both cases additional beams form besides the ones injected by the colliding pulses. This is the result of another injection scheme called phase kick injection, details of which can be found elsewhere [29].

While such hybrid simulations are possible, we have found that they do not yet work well for highly asymmetric cells, as are required for laser pulses containing a large number of wavelengths. Asymmetric cells are known to be problematic in fluid numerics. Thus, the use of hybrid simulations for extreme cases appears to require algorithm development.

8. Summary and future directions

Using object-oriented programming and other modern computing techniques, we have developed a highly flexible plasma simulation code, VORPAL. VORPAL can incorporate multiple implementations for the electromagnetic field. For the plasma there are both fluid and particle implementations. In addition, VORPAL can be run in any dimensionality, with the dimension chosen at run time. VORPAL also has a flexible domain decomposition into an arbitrary set of slabs. VORPAL is built with the GNU configure tools, and, thus, it runs on a variety of platforms.

VORPAL is now in use as a plasma simulation code with multiple implementations. The electromagnetic field can be an FDTD field on a Yee mesh, a prescribed field, or any combination of those. Particle implementations with relativistic, non-relativistic, and electric force only are available. Two fluid implementations are available.

The various implementations in VORPAL have been validated. It was shown to give the correct results for plasma oscillations. In hybrid mode VORPAL was shown to correctly produce the wake field generated by a laser pulse propagating in a plasma, and it was capable of simulating beam injection via colliding laser pulses.

A variety of future projects are possible with VORPAL. The addition of an implicit Maxwell solver [30] would allow simulation of systems in which light waves are not important. An implicit fluid solver could remove the electron plasma oscillations. In combination, these directions should allow VORPAL to be used to model low-frequency plasma phenomenon such as RF heating of fusion plasmas. The addition of complex boundary conditions would allow studies of microwave cavities, and the addition of ionization would make it a strong plasma processing code. Studies of beam cooling using a fast multipole method for the fields have already begun.

Acknowledgements

The authors would like to thank J.P. Verboncoeur, D.L. Bruhwiler, B.A. Shadwick, R. Busby, P. Messmer, R. Giacone, J. Regele, P.J. Mardahl, P.H. Stoltz and V. Przebinda for helpful discussions and contributions to code development. The parallel scaling tests and some parallel code development was done at the National Energy Research Scientific Computing center (NERSC). This work was supported by the National Science Foundation Grant PHY-0112907 and the Department of Energy Grant DE-FG03-95ER54297.

References

[1] C.K. Birdsall, A.B. Langdon, Plasma Physics Via Computer Simulation, Adam Hilger, 1991.
[2] R.W. Hockney, J.W. Eastwood, Computer Simulation Using Particles, Adam Hilger, 1988.
[3] J.P. Verboncoeur, A.B. Langdon, N.T. Gladd, An object-oriented electromagnetic PIC code, Comp. Phys. Commun. 87 (1995) 199.
[4] D.L. Bruhwiler et al., Particle-in-cell simulations of plasma accelerators and electron-neutral collisions, Phys. Rev. ST Accel. Beams 4 (2001) 101302.
[5] R.A. Fonseca, L.O. Silva, R.G. Hemker, F.S. Tsung, V.K Decyk, W. Lu, C. Ren, W.B. Mori, S. Deng, S. Lee, T. Katsouleas, J.C. Adam, OSIRIS: a three-dimensional, fully relativistic, particle in cell code for modeling plasma based accelerators, in: P.M.A. Sloot, C.J.K. Tan, J.J. Dongarra, A.G. Hoekstra (Eds.), Computational Science – ICCS 2002, Part III, Lecture Notes in Computational Science, vol. 2331, Springer, 2002, p. 342.
[6] J. Wang, P.C. Liewer, V.K. Decyk, 3D electromagnetic plasma particle simulations on a MIMD parallel computer, Comput. Phys. Commun. 87 (1995) 35.
[7] Tech-X. Available from: <http://www.techxhome.com/products/optsolve/index.html>, accessed Dec. 2002, 1998.

[8] R. Robson, Using the STL: The C++ Standard Template Library, Springer, 1998.
[9] T. Tajima, J.M. Dawson, Laser electron-accelerator, Phys. Rev. Lett. 43 (1979) 267.
[10] R. Giacone et al., Generation of nonlinear plasma wake fields in the colliding laser pulses injection schemes, Bull. Am. Phys. Soc. 47 (2002) 282.
[11] J.R. Cary, C. Nieter, Available from: <http://www-beams.colorado.edu/vorpal/>, accessed, June 2003, 2002.
[12] HDF5, a new generation of HDF, Available from: <http://hdf.ncsa.uiuc.edu/HDF5> (last accessed Dec. 2002.
[13] Y. Pemper, B. Fidel, E. Heymana, R. Kastner, R.W. Ziolkowski, Study of absorbing boundary conditions in the context of the hybrid ray-FDTD moving window solution, in: The Thirteen Annual Review of Progress in Applied Computational electromagnetics, ACES Symposium, 1997, p. 17.
[14] C.D. Decker, W.B. Mori, Group-velocity of large-amplitude electromagnetic-waves in a plasma, Phys. Rev. Lett. 72 (1994) 490.
[15] J. Villaseñor, O. Buneman, Rigorous charge conservation for local electromagnetic-field solvers, Comput. Phys. Commun. 69 (1992) 306.
[16] W. Gropp, E. Lusk, A. Skjellum, Using MPI, The MIT Press, 1999.
[17] P.M. Campbell, E.A. Carmona, D.W. Walker, Hierarchical domain decomposition with unitary load balancing for electromagnetic particle-in-cell codes, in: D.W. Walker, Q.F. Stout (Eds.), Proceedings of the 5th Distributed Memory Conference, IEEE Computer Society Press, 1990, p. 943.
[18] M.J. Berger, S.H. Bokhari, A partitioning strategy for nonuniform problems on multiprocessors, IEEE Trans. Comput. C-36 (5) (1987) 570.
[19] S.B. Baden, Programming abstractions for dynamically partitioning and coordinating localized scientific calculations running on multiprocessors, SIAM J. Sci. Stat. Comput. 12 (1) (1991) 145.
[20] E.A. Carmona, L.J. Chandler, On parallel PIC versatility and the structure of parallel PIC approaches, Concurrency 9 (12) (1997) 1377.
[21] G. Sasser, J. Hanvranek, S. Colella, J. Luginsland, L. Kerkle, Modified PIC algorithm of efficient multiprocessor simulations, in: Proceedings of the 16th International Conference on Numerical Simulation of Plasmas, 1998, p. 151.
[22] P.C. Liewer, V.K. Decyk, A general concurrent algorithm for plasma particle-in-cell simulation codes, J. Comput. Phys. 85 (1989) 302.
[23] J.D. Blahovec, L.A. Bowers, J.W. Luginsland, G.E. Sasser, J.J. Watrous, 3-D ICEPIC simulations of the relativistic klystron oscillator, IEEE Trans. Plasma Sci. 28 (3) (2000) 821.
[24] K.S. Yee, Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media, IEEE Trans. Antennas Propagat. 14 (1966) 302.
[25] E.S. Oran, J.P. Boris, Numerical Simulation of Reactive Flow, Cambridge University Press, 2001.
[26] S.T. Zalesak, Fully multidimensional flux-corrected transport algorithms for fluids, J. Comput. Phys. 31 (1979) 335.
[27] J.P. Boris, D.L. Book, Flux-corrected transport. 1. SHASTA, a fluid transport algorithm that works, J. Comput. Phys. 11 (1973) 38.
[28] E. Esarey, R.F. Hubbard, W.P. Leemans, A. Ting, P. Sprangle, Electron injection into plasma wake fields by colliding laser pulses, Phys. Rev. Lett. 79 (1997) 2682.
[29] J.R. Cary, R. Giacone, C. Nieter, D.L. Bruhwiler, E.H. Esarey, W.P. Leemans, All-Optical Beamlet Train Generation, in: Proceedings of the 2003 Particle Acclerator Conference, 2003.
[30] K.J. Bowers, Implicit methods of solving the maxwell equations suitable for particle-in-cell simulations of low temperature plasmas, J. Comput. Phys. (2001), submitted.