New Methods in WARP

D. P. Grote, Alex Friedman, LLNL L-645, PO Box 808, Livermore, CA 94550 and
I. Haber, NRL, Washington DC 20375-5346[*]

Abstract

The WARP[1] code is being developed and applied to simulate the creation and propagation of the high-current, space-charge dominated beams that are required for heavy-ion driven fusion energy (HIF). New methods and capabilities have recently been introduced into WARP, a multi-dimensional particle-in-cell code developed for the study of space-charge dominated beams. We describe: (a) a 2D3V "slice" model (WARPxy) with two novel capabilities: the optional use of 3D applied fields (which can be calculated using the WARP3d solver), and an "exact" treatment of a bent beam pipe, via coordinate transformations; (b) a multigrid fieldsolver which offers internal conductors in 2D and 3D; (c) serial optimizations for cache-based machines which yielded a 20-30% speedup; and (d) a Python interface which has been developed to give the full power of a scripting language user interface in both serial and parallel computing environments.

1 INTRODUCTION

A heavy-ion induction accelerator is a promising candidate for a driver for inertial confinement fusion power production. In order to drive the target to ignition, a driver must produce a beam with a high current (several kAs), moderate energy (several GeVs), but very low emittance (several π-mm-mrads). The beams are space-charge dominated and behave like non-neutral plasmas. In order to achieve the required emittance, a thorough understanding is needed of the behavior of the beam and the effects of manipulations and errors. We have developed WARP, a multi-dimensional particle-in-cell/accelerator code, to study the physics of such high-current, low emittance beams.

WARP has been designed and developed to be a flexible simulation tool, allowing simulation of all sections of a driver at various levels of detail and dimensionality. Some of the recently implemented methods and capabilities which enhance that flexibility are described here: new physics models, computational techniques, and computer science issues.

2 SLICE MODEL

A slice model is a transverse model of a beam, ignoring some longitudinal effects, primarily longitudinal variations in the space-charge fields. The slice model can be understood as a model of the behavior of the central portion of the beam (where the beam is fairly uniform axially) versus time, or as the behavior of an infinitely long beam as a function of distance. Slice simulations are not new, in fact the first simulations of beams for heavy-ion fusion were slice simulations, but the model implemented in WARPxy contains several new important features.

The slice code WARPxy was originally adapted from, and is closely coupled to, the WARP3d code, immediately giving it the full power of problem specification and diagnostics of the three-dimensional code. An example is a rich set of methods for specifying the fields of accelerator lattice elements in WARP3d that was easily adapted to work within the slice code. The close coupling allows much sharing of coding which is common to the two models, such as particle moments calculations and other diagnostics.

An important issue in beam dynamics is the presence of an axial velocity spread, which can lead to an increase in transverse emittance and directly affects the final spot size on target. A scheme was adopted in the slice code to include the axial velocity spread as well as changes in the axial velocity, such as from acceleration gaps and from axial force components of other lattice elements. Another issue is the effect of bends on the beam. It is known that when a beam with an axial velocity spread enters a bend, its emittance will increase[2]. Bends are present in the driver designs to guide the beams into the target chamber, and more significantly, in driver designs based on recirculating accelerators. The same scheme adopted for the velocity spread was adapted to include bends, where the time step size of each particle is a function of its radial position.

The underlying integration method for advancing the particles is the leap-frog method. In the slice model, the step size has a constant physical length, Δs, and so the time step size is dependent on the axial velocity of the individual particles. When the axial velocity of the particles changes, the time step-size must be adjusted. The algorithm used in WARPxy is to iterate over the first two steps of the split-leap frog time advance.

$$\vec{v}^{n+\frac{1}{2}} = \vec{v}^n + \frac{1}{2}\frac{\vec{F}^n}{m}\Delta t \qquad (1)$$

$$\vec{x}^{n+1} = \vec{x}^n + \vec{v}^{n+\frac{1}{2}}\Delta t \qquad (2)$$

Here $\vec{x}$ and $\vec{v}$ are the position and velocity, $\vec{F}$ is the force, including electric and magnetic fields, and n is the time level and the step size is Δt. After this partial advance, the time-step size is scaled by the amount of over- or under-

[*] Word performed under the auspices of the U.S. D.O.E. by Lawrence Livermore National Laboratory and Naval Research Laboratory under contracts W-7405-ENG-48, DE-AI02-93ER40799, and DE-AI02-94ER54232.

advance of the axial position, z.

$$\Delta t' = \Delta t \frac{\Delta s}{z^{n+1} - z^n} \tag{3}$$

The partial advance is then redone with the new time-step size. Once the iteration converges and the correct time-step size is obtained, the particle advance is completed.

$$\vec{v}^{n+1} = \vec{v}^{n+\frac{1}{2}} + \frac{1}{2}\frac{\vec{F}^{n+1}}{m}\Delta t' \tag{4}$$

Our experience is that the iteration converges very rapidly. The code always iterates all of the particles a set number of times, which can be controlled by the user.

When in a bend, the same iteration is done but with the scaling of the time-step size also accounting for the rotation of the slice frame around the bend. The scaling is done in polar coordinates relative to the center of the bend.

$$\Delta t' = \Delta t \frac{\Delta \theta}{\theta^{n+1} - \theta^n} \tag{5}$$

Here, $\Delta\theta = \Delta s / r_{\rm bend}$ where $r_{\rm bend}$ is the bend radius, and θ^n is the axial position of the particle in the polar coordinate system at time level n. This iteration converges as rapidly as with no bends.

The calculation of the beam self-fields must also include the curvature of the bends. When the self-fields are calculated by solving Poisson's equation, the curvature terms can be treated as source terms, iterating to reach convergence. While alternative direct methods are available to solver Poisson's equation with the curvature terms, the iterative method was chosen since the same method is used in WARP3d. Also, the speed of the field solution is not critical since the two-dimensional simulation time tends to be dominated by the time of the particle advance.

The algorithm has been thoroughly tested. Numerous single particle tests have shown empirically that the method is second order in Δt, the same as the underlying leap-frog advance. Full beam tests have also be carried out. Two tests are described here.

As a test of the bending algorithm, a beam is propagated through a straight lattice but with the coordinate system of the simulation following s-bends. The s-bends deviate from the beam centroid path by roughly the beam radius. The simulated beam behaved as expected, reproducing the correct envelope and showing no anomalous growth in the emittance.

In another test, the behavior of a beam with an axial velocity spread as it entered a bend was examined. The slice simulations were compared with both the three-dimensional simulation and an analytic theory developed to explain the emittance growth in a bend[2]. Good agreement was found among all three. The slice simulations show oscillations in the emittance which agree in amplitude and frequency with the WARP3d simulation and theory. The simulations do show damping in the emittance oscillations that is not included in the theory. The analytic theory does give asymptotic limits on the emittance growth and these agree with the simulation results.

3 MULTIGRID POISSON SOLVER

The WARP code uses an electrostatic model of the self-field of the beams which requires solving Poisson's equation to calculate the potential on a Cartesian mesh. The code now has three primary Poisson solvers. The fastest is an FFT based solver which does sine-sine-periodic FFT's to model an infinitely long conducting square pipe. Capacity matrix methods can be used to include simple boundaries such as a round conducting pipe. With more complex conductor geometry, however, the required matrix rapidly becomes too large, and so an iterative method, successive over-relaxation (SOR), was implemented for such cases. The SOR method allows inclusion of arbitrarily complex conductor geometry with much less penalty. Unfortunately though, for the typical mesh sizes used, the SOR method is roughly ten times slower than the FFT method without any conductors, and becomes comparatively worse with larger mesh sizes. In the three-dimensional simulations using the SOR method, the total simulation time was dominated by the Poisson solve. Because of the computational time required to generate large capacity matrices or to use the SOR method, the grid resolution and propagation distance has been limited when using complex conductor geometry.

A third method, the multigrid method[3], has recently been implemented. It promises faster solution while having a small penalty for complex conductor geometry. The multigrid method for solving Poisson's equation on a Cartesian mesh was adapted to include internal conductors. The SOR solver is used to iteratively refine the error on each of the coarse grid levels, allowing use of the same techniques for applying internal conductors which were developed for the full SOR solver. The two techniques used are forcing of the potential inside of the conductor to the desired value each iteration, and a subgrid-scale resolution method in which the finite difference form of Poisson's equation is modified for points within one grid cell outside the surface of the conductors to explicitly include the location of the surface[4].

In order to achieve the rapid convergence of the multigrid method, the conducting boundary conditions must be applied at all levels of coarseness. So, for each level, a list of the grid points which are affected by a conductor is required. For the points inside of the conductor, this offers no difficulty since only a simple check is needed to determine whether the points on the finest grid that are inside of a conductor are on the coarse grid.

The subgrid-scale resolution technique, though, requires additional data. On the coarse grid levels, there will be points within one coarse grid cell of the conductor surface that were more grid cells away on the finest grid. Those points must be included. To generate this data, an algorithm was developed that begins with the subgrid-scale data for the finest grid and scans the grid at each of the coarse levels to gather the data for the coarse levels. For each of the points in the total list, the lowest coarseness level at which a point is on a grid is saved. So, at each level of coarseness, a point must pass two tests to determine if it is near a conductor: the coarseness level at which the point is on a grid must be less than or equal to the current coarseness level, and the point must be within one grid cell of a conductor.

The multigrid method has the same scaling of operation count with the number of grid cells as the FFT. Timings, see Table 1, show that the FFT is about three to four times faster for all of the grid sizes examined. While the inclusion of conductors does slow the solver down, the solve times are still far below that of the original SOR method. The increase in speed will allow simulations with higher resolution and longer propagation distances than was practical with the original SOR solver.

Table 1. Timings of the field solvers on a single process of a Cray J90. All times are in seconds. The numbers in parenthesis are the number of iterations required to reach the desired convergence.

	Multigrid	FFT	SOR
64x64x64	1.6 (8)	0.41	4.7 (170)
128x128x128	10.6 (8)	2.9	74.0 (340)
ratio of timings 128/64	6.6	7.1	15.7
64x64x64 with conductors	8.7 (14)	-	17.4 (230)

4 OPTIMIZATION TECHNIQUES

Since WARP3d is a three-dimensional code, the simulation time can grow very large. Because of this, much effort has been put into optimizing the code. The optimization can be done by using advanced computational methods to reduce the total amount of computation needed, as with the implementation of the multigrid solver described above, or as with the use of higher-order integration methods[5]. The methods discussed here are methods for increasing the speed at which a set amount of computation is done.

There are a number of simple optimizations that can be done which reduce the time of computation on serial machines. For example: combine constants in loops and remove them completely from loops if possible; remove divides from loops if possible, and if not possible, put the divide as early as possible if the result will be used inside of the loop; treat multidimensional arrays as one-dimensional arrays; use “if” statements to avoid unnecessary work, even inside of loops; and optimize the cache use. The last technique is described more fully.

Most modern day computers have non-uniform memory access: many more CPU cycles are required to bring data from the main memory to the cache, than from cache to the register where the CPU can use the data. The difference is as large as a factor of thirty! The goal, then, is to reuse data as much as possible once it is in the cache. There are three basic techniques. The first is to take advantage of the fact that the data is brought into the cache in chunks. Grouping data together in memory that is used together in loops or making inner loops over the first indices of arrays (in Fortran) accomplishes this. The second technique is to maximize the number of mathematical operations for each fetch/store from memory. Combining loops is a good way to accomplish this. An implication of this is that the array syntax of Fortran 90 will be bad for cache reuse since it tends to break calculations into shorter loops. The third method relies on knowing the cache line size, the size of the chunk of data that is brought into the cache at once. The arrays should be arranged in memory so that multiple arrays used inside of the same loop do not push each other out of the cache.

The 3-D FFT Poisson solver is used as an example of where cache reuse can improve the execution time. The FFT’s are organized as “gang” FFT’s - in the transforms over each dimension, the inner loops in the transform are over one of the other dimensions. For the transverse transforms, the outer loop is over the third dimension, while the first two dimensions switch roles between the transformed and the ganged. In both cases, the data is well localized. For the transform over the third dimension, though, the data is not localized, there is a large stride in memory, and there is little cache reuse. Timings of the code bear this out; the transform over the third dimension runs several times slower than the transforms over the first two (which are roughly equal).

The way around the decreased performance is to temporarily rearrange the data to localize it in memory for the transform. The 3-D data is divided into slices along the second dimension, each slice consisting of a plane of data with the first dimension as one axis and the third dimension as the other. Each slice is copied into a temporary array and then transformed along the third dimension. This gives the same locality as in the original transverse transform. Afterward, the data is copied back. The resulting transform time is reduced to be same as the time of the transverse transform. One of the copies is “free” since the array was looped over anyway to multiply it by a constant and so the penalty is the one additional copy. The gain however is a factor of several reduction in the transform time for that dimension. The total gain over the full Poisson solve is typically a 30% reduction in computation time.

5 PYTHON INTERFACE TO WARP

Our experience has shown that to realize the full power and capability of a large code, a high-level, flexible, sophisticated interface is required. A primary requirement of the interface is full language support for user-programmable code control that can be used interactively. An interpreted scripting language provides such an interface. Another required element is visualization - for pre- and post-processing as well as for interactive use. In this section we describe the work we have done to maintain such a scripting language interface in both the serial and parallel computing environments.

5.1 *Why use an interpreter interface?*

An interpreter interface provides a much more flexible means of inputting data and controlling a code than the more traditional command-line arguments and namelist style interface. It does this by allowing the use of high-level language constructs and by giving access to the run-time database and functionality of the code. This makes it easier to design a system where the code developers create a set a packages that users can combine and control to suit theirs needs, rather than create a code where the developers build in a limited number of options that the users must choose from.

An interpreter allows for more rapid code development. Use of an interpreter removes the need for compile and load steps, directly reducing development time. It also acts as a built in debugger, giving full access to the data, but is more powerful, allowing independent testing of code segments. Once algorithms have been developed and tested, they can be converted into compiled code if there is a problem with speed.

The code size is reduced when an interpreter is used. Most interpreters provide extensive services such as graphics, memory management, and data dump and restart facilities, for example. These are accessible from the interpreted language and do not need to be referenced from the compiled portion of the code. Only the core routines and algorithms need to built into the code, the rest can be written at the interpreter level. Special features which have a one time use or are required by certain users, for example, can be kept out of the main code and written in the interpreted language.

An interpreter can acts as "glue," linking different codes and packages together. The packages can have separate variable name-spaces and can be developed separately by different developers in different languages. The packages are brought together at the interpreter level; all have the same interface.

WARP was originally built with the Basis code development/interpreter system that was developed at LLNL[6]. Basis has a number of advantages over other interpreters. It has a built in mechanism for generating the interface between the compile code and the interpreter language. It also has a number of other important features built in, such as graphics, memory management, data dumps and restarts. Also, the language is based on Fortran, making it easy to use and convert into compiled Fortran. Another important advantage is the modularity. Basis allows the code the be developed as a set of independent packages with separate name-spaces. The packages are glued together at the interpreter.

There are a number of disadvantages of Basis as well. It is a rather large system which is not very portable. Basis runs only on Unix and Linux machines. It is missing some programming features, such as structures and objects. Also, it has a small number of developers, limiting its breadth of features and development.

5.2 *Parallel WARP with Basis*

In our first attempt at a user interface to parallel version of WARP, we wanted to retain Basis. Unfortunately, Basis has not been ported to the MPP architectures we use. We were able to get around this by running Basis on a local serial workstation and having it spawn and control processes on a remote MPP. Communication was done with PVM[7] since it allowed spawning of processes. The processes on the MPP were event driven - they would wait for commands that the user sends via the Basis interpreter and PVM.

While this system was effective and allowed us to get WARP up and running on an MPP, it had significant drawbacks. Since there was no interpreter running on the MPP side, all possible desired commands had to be preprogrammed, so access to runtime database and controllers was limited to those for which special routines had been written. Because of the requirement for spawning, the code could only be run interactively - batch jobs could not be run. Also, the system required a "close connection" between the workstation and the MPP. Due to computer center policy restrictions, this is not always possible. Because of this last drawback, the parallel version of WARP was in fact inoperable on the primary MPP machine we had access to.

5.3 *Python*

We needed to make use of another interpreter and decided on Python[8]. Python is a recently developed interpreter to which compiled code can be linked and which has a number of advantages over Basis. The language is fully object-oriented (supporting inheritance and polymorphism). It is also small and portable and runs on almost all types of machines, UNIX, PC's running Linux or windows, and Macintoshes. Python also has a large number of developers and has a broad base of available packages, such as linear algebra libraries, interfaces to graphical user interface development libraries, world-wide-web software, and notably, a parallel adaptation.

There are problems with Python, though. A minor problem is the language itself, which has attributes which may be unfamiliar to many scientific users, as discussed below. Python also lacks standards for important features such as graphics and data dumps. There is also no built in method of automatically generating the interface between the compiled code and the interpreter language. While there are packages that do this, they are not flexible enough to meet all of our needs. Note that while the interface can be created by hand, doing so for a large code is not practical considering the size of the interface and the constant need for updating it.

We had to develop our own software to automatically generate the interface between Fortran and Python. The software takes advantage of the work which was done for the Basis interface. The same variable description files were used - a parser was written in Python to extract the information needed from the files to create the interface. The

Fortran preprocessor used with the Basis system is kept so that the original compiled code can be used essentially unchanged. The coarse-grained object-oriented nature of the Basis code is maintained. Each of the original Basis packages is turned into a Python object and has its own namespace.

The most important detail of the interface is the way compiled variables are accessed from the interpreter. In the Python language, all variables are references to objects. The major implication of this is that an assignment, such as "a = b", is actually a re-reference. Before the assignment, the variables "a" and "b" refer to different objects. After the assignment, "a" and "b" refer to the same object, that to which "b" originally referred, and the object to which "a" referred is lost. This means that there cannot be a direct connection between a Python variable and a Fortran variable, since that connection would be lost on an assignment.

The way around the re-referencing is to make use of "attributes" of objects, which are like class members and functions. The getting and setting of attributes can be redefined, allowing the possibility of connecting an attribute to a compiled variable or function via the get and set routines. Get and set routines are defined which perform a search through the list of Fortran variables and subroutines of a package object to find the one associated with the Python attribute.

A generic Python type is created and each package is defined as being an object of that generic type. The definition of the type includes generic functions, such as the get and set, which take the package object as an argument. In creating an object for a package, all of the required information is stored, such as the list of Fortran variables and subroutines.

5.4 Parallel WARP with Python

Python is easily ported to an MPP environment, with one caveat: the input and output must be controlled. Software written by others[9] was obtained and used to do this. This software is designed so that only one processor can read the user input. That process then passes the input to the other processes via message passing. The user can control which processes can print output - the default is only the process that reads the input.

Combining the existing parallelized Fortran code with the Python interface and the input/output package gives a fully interpreter-driven code on the MPP which has the same user interface as the serial code. This combined system is nearly fully operational; nearly all of the functionality which was available with the original serial Basis version is now available with the Python version, both serial and parallel.

The development of the Python interface was a significant step in the evolution of WARP. Since Basis is used in several important LLNL codes, it will likely be around and supported for a long time. However, some of its disadvantages are unlikely to be removed. These include a lack of structures and objects, limited available software packages, and a lack of portability. For these reasons, we are currently retaining both versions (the source is the same, only the interface and scripts are different). We are converting the Basis scripts to Python as they are needed.

6 CONCLUSIONS

While WARP is in some ways a mature code, it is still rapidly evolving. We are adding new physics models, such as the WARPxy model described. We are developing more advanced computational techniques, such as the multigrid method for solving Poisson's equation. We are also adopting modern computer science techniques, including optimization through cache reuse, use of the modern scripting language Python for code steering and user programmability, and massively parallel computation via message passing.

7 REFERENCES

[1] D. P. Grote, A, Friedman, I. Haber, S. Yu, "Three-Dimensional Simulations of High-Current Beams in Induction Accelerators with WARP3d", *Proceedings of the 1995 International Symposium on Heavy Ion fusion, Fusion Engineering and Design*, 32-33 (1996) 193-200.

[2] J. J. Barnard, H. D. Shay, S. S. Yu, A. Friedman, and D. P. Grote, "Emittance growth in Heavy Ion Recirculators", 1992 Linear Accelerator Conference Proceedings, Ottawa, Canada, vol. 1, p 229, AECL Research, (1992).

[3] William H. Press et. al., *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, 1986.

[4] D. P. Grote, A. Friedman, I. Haber, "Methods used in WARP3d, a Three-Dimensional PIC/Accelerator Code", *Proceedings of the 1996 Computational Accelerator Physics Conference*, AIP Conference Proceedings 391, p. 51.

[5] A. Friedman, D. P. Grote, I. Haber, "Towards Higher Order Particle Simulation of Space-Charge-Dominated Beams", *Proceedings of the 16th International Conference on the Numerical Simulation of Plasmas*, February 10-12, 1998, Santa Barbara, CA.

[6] P. F. Dubois, The Basis System, LLNL Document M-225 (1988)

[7] Al Giest, et. al., "PVM3 User's Guide and Reference Manual", technical report ORNL/TM-12187.

[8] The best reference is the Web page http://www.python.org/

[9] The primary authors are D. Beazley and T. B. Yang.