Journal of Computational Physics **163,** 434–451 (2000)
doi:10.1006/jcph.2000.6570, available online at http://www.idealibrary.com on IDEAL®

An Object-Oriented Parallel Particle-in-Cell Code for Beam Dynamics Simulation in Linear Accelerators

Ji Qiang,* Robert D. Ryne,† Salman Habib,† and Viktor Decyk‡

**Ms H817, LANSCE-1, and †Ms B288, T-8, Los Alamos National Laboratory, Los Alamos, New Mexico 87545; and ‡Physics Department, University of California at Los Angeles, Los Angeles, California 90024*
E-mail: jiqiang@lanl.gov, ryne@lanl.gov, habib@lanl.gov, vdecyk@pepper.physics.ucla.edu

Received June 6, 1999; revised June 13, 2000

We present an object-oriented three-dimensional parallel particle-in-cell (PIC) code for simulation of beam dynamics in linear accelerators (linacs). An important feature of this code is the use of split-operator methods to integrate single-particle magnetic optics techniques with parallel PIC techniques. By choosing a splitting scheme that separates the self-fields from the complicated externally applied fields, we are able to utilize a large step size and still retain high accuracy. The method employed is symplectic and can be generalized to arbitrarily high order accuracy if desired. A two-dimensional parallel domain decomposition approach is employed within a message-passing programming paradigm along with a dynamic load balancing scheme. Performance tests on an SGI/Cray T3E-900 and an SGI Origin 2000 show good scalability of the object-oriented code. We present, as an example, a simulation of high current beam transport in the accelerator production of tritium (APT) linac design. © 2000 Academic Press

Key Words: object-oriented; particle-in-cell; beam dynamics; linear accelerators.

I. INTRODUCTION

The simulation and analysis of charged particle beam transport is an important subject in accelerator design and optimization. The increasing interest in high-intensity beams for future accelerator applications presents challenging problems that require one to understand and predict the dynamics of beams subject to complicated external focusing and accelerating fields, as well as the self-fields caused by Coulomb interaction of the particles. One approach to studying the behavior of these particles is to use envelope equations [1–3]. These equations are a set of ordinary differential equations for the second-order moments of the particle distribution and can be calculated quickly. However, the envelope equations do not provide a detailed description of the beam. Furthermore, integration of the envelope equations requires information about certain aspects of the beam's evolution (such as the beam emittance) that is not generally known *a priori*. Future accelerator applications place extremely stringent requirements on particle loss, which is associated with the low-density, large-amplitude halo of the beam. The need to model the details of the beam distribution, in the presence of strong self-fields, leads us to a full Poisson–Vlasov description to better understand and predict the behavior of intense charged particle beams in accelerators.

The Poisson–Vlasov equations can be solved using a phase space grid-based method or a PIC method. The grid-based method is effective in one and two dimensions [4]; but for three-dimensional systems with six phase space variables, the grid-based method will require an enormous amount of memory even for a coarse grid. Also, grid-based methods may break down when very-small-scale structures form in the phase space. The PIC method has a much lower storage requirement and will not break down even when the phase space structure falls below the grid resolution. This method is widely used to study the dynamics of high-intensity beams in accelerators [5–9].

The computational time cost associated with using a large number of numerical particles restricts that number and limits the accuracy of PIC calculation on serial computers. The parallel PIC method, which was developed largely by the plasma physics community and, to a lesser extent, by the astrophysics community, has made it possible to perform large-scale PIC simulations on multiprocessor platforms [10–17]. The parallel PIC approach provides a means of reducing fluctuations by enabling use of more particles and of improving spatial resolution through increased grid density. It also dramatically reduces the computation time. However, except in a few cases [7, 18], the parallel PIC approach has not been widely used in the accelerator community.

In this paper, we present an object-oriented parallel PIC code for beam dynamics simulation in linear accelerators (linacs). The object-oriented approach gives the program good maintainability, reusability, and extensibility. In addition to describing the object-oriented implementation on parallel computers, we will describe the use of split-operator methods, which provide a powerful means of including space-charge effects in single-particle beam transport codes. The result is a multiparticle capability that combines sophisticated techniques of magnetic optics with those of parallel PIC simulation.

The organization of this paper is the following: The physical model and numerical methods are described in Section II. The parallel PIC algorithm using MPI on distributed parallel machines is discussed in Section III. The object-oriented software design for beam dynamics simulation is given in Section IV. Performance tests are given in Section V. An application of the code to a simulation of the APT linac design is presented in Section VI. The conclusions are drawn in Section VII.

II. PHYSICAL MODEL AND NUMERICAL METHODS

The equations governing the motion of individual particles in an accelerator (in the absence of radiation) are Hamilton's equations,

$$\frac{d\mathbf{q}}{dt} = \frac{\partial H}{\partial \mathbf{p}} \qquad \frac{d\mathbf{p}}{dt} = -\frac{\partial H}{\partial \mathbf{q}}, \tag{1}$$

where $H(\mathbf{q}, \mathbf{p}, t)$ denotes the Hamiltonian of the system, and where $\mathbf{q}$ and $\mathbf{p}$ denote canonical coordinates and momenta, respectively. Let ζ denote the six-vector of coordinates and momenta. In the language of *mappings*, which are a major theme in modern accelerator physics, we would say that there is a (generally nonlinear) map, $\mathcal{M}$, corresponding to the Hamiltonian, H, which maps initial phase space variables, ζ^{i}, into final variables, ζ^{f}, and we write

$$\zeta^{\mathrm{f}} = \mathcal{M}\zeta^{\mathrm{i}}. \tag{2}$$

Sophisticated techniques now exist for computing maps corresponding to externally applied electromagnetic fields to essentially any order, for combining and manipulating maps, for applying maps to phase space coordinates (or functions of the coordinates), and for analyzing maps [19]. Note that Hamilton's equations can be rewritten as

$$\frac{d\zeta}{dt} = -[H, \zeta], \tag{3}$$

where [,] denotes the Poisson bracket. The corresponding equation governing the beam distribution function, $f(\zeta, t)$, is simply the Liouville equation,

$$\frac{df}{dt} = \frac{\partial f}{\partial t} - [H, f] = 0, \tag{4}$$

from which we obtain

$$\frac{\partial f}{\partial t} = [H, f]. \tag{5}$$

It is straightforward to show that the evolution of a distribution function (i.e., the solution of Eq. (5)) is also contained in $\mathcal{M}$. Namely, a distribution function $f(\zeta, t)$ whose initial value is $f^0(\zeta) = f(\zeta, 0)$ evolves according to

$$f(\zeta, t) = f^0(\mathcal{M}^{-1}\zeta). \tag{6}$$

So far we have implicitly assumed that we are dealing with particles subject only to externally applied fields. We can treat the dynamics in the presence of external fields and self-fields (i.e., space-charge fields) by including them both in the single-particle Hamiltonian. In many cases one can write

$$H = H_{\mathrm{ext}} + H_{\mathrm{sc}}, \tag{7}$$

where H_{ext} denotes the Hamiltonian in the absence of self-fields and H_{sc} denotes the Hamiltonian associated with the space-charge fields. In accelerator physics, H_{ext} is often an extremely complicated function, perhaps containing hundreds of thousands of terms. This is due to the fact that it normally involves Taylor expansion of the Hamiltonian in order to perform high-order perturbation theory around a reference trajectory. In contrast with the treatment of external fields, self-fields governed by H_{sc} are not normally treated as power series in the canonical variables because the variation over the domain of interest is too great. In many cases H_{sc} is simply proportional to the scalar potential ϕ, which satisfies the Poisson equation,

$$\nabla^2\phi(\mathbf{q}) = -\rho(\mathbf{q}). \tag{8}$$

The combination of Eqs. (5) and (8) constitutes the Poisson–Vlasov system of equations.

Our approach to solving these equations involves the use of split-operator methods. As just mentioned, beam dynamics calculations often involve a Hamiltonian that can be written as a sum of two parts, $H = H_{\text{ext}} + H_{\text{sc}}$. Such a form is ideally suited for the application of symplectic split-operator methods [20]. More generally, consider a Hamiltonian that can be written as a sum of two parts, $H = H_1 + H_2$, where each part, separately, can be solved exactly or to some desired accuracy or order. In other words, suppose that we can compute the mapping $\mathcal{M}_1$ corresponding to H_1 and the mapping $\mathcal{M}_2$ corresponding to H_2. In our case H_1 includes the external fields, and $\mathcal{M}_1$ can be computed to any order using the techniques of magnetic optics; the second term, H_2, corresponds to the space-charge fields, and can be dealt with using parallel particle simulation techniques. Given $\mathcal{M}_1$ and $\mathcal{M}_2$, the following algorithm is accurate through second order in τ,

$$\mathcal{M}(\tau) = \mathcal{M}_1(\tau/2)\ \mathcal{M}_2(\tau)\ \mathcal{M}_1(\tau/2)\ , \tag{9}$$

where τ denotes the time step. (In accelerator physics, one often uses a coordinate as the independent variable. However, for the sake of this discussion we will continue to refer to τ as a time step.) The second-order accuracy of this algorithm is easily demonstrated using Lie methods [21, 22]. From Eq. (3), we can write the formal solution of Hamilton's equations as

$$\zeta(\tau) = \exp(-\tau : H :)\,\zeta(0). \tag{10}$$

Here, we have defined a differential operator $: H :$ as $: H : g = [H, g]$, for arbitrary function g. For a Hamiltonian that can be written as a sum of two parts $H = H_1 + H_2$, we have the formal solution

$$\zeta(\tau) = \exp(-\tau(: H_1 : + : H_2 :))\,\zeta(0). \tag{11}$$

For noncommutative operators X and Y, the Campbell–Baker–Hausdorff formula states that

$$\exp(X)\exp(Y) = \exp\left(X + Y + \frac{1}{2}\{X, Y\} + \frac{1}{12}(\{X, \{X, Y\}\} + \{\{Y, X\}, Y\}) + \cdots\right), \tag{12}$$

where the $\{X, Y\}$ denotes $XY - YX$ [23–25]. By repeated application of the Campbell–Baker–Hausdorff formula, we obtain

$$\begin{aligned} &\exp(-\tau(: H_1 : + : H_2 :)) \\ &\quad = \exp\left(-\frac{1}{2}\tau : H_1 :\right)\exp(-\tau : H_2 :)\exp\left(-\frac{1}{2}\tau : H_1 :\right) + O(\tau^3), \end{aligned} \tag{13}$$

where $\exp(-\frac{1}{2}\tau : H_1 :)$ defines map $\mathcal{M}_1$ and $\exp(-\tau : H_2 :)$ defines map $\mathcal{M}_2$. Therefore, Eq. (9) has second-order accuracy, as stated.

As a side note, if we were to use a different splitting in which H_1 depended only on momenta and H_2 depended only on positions, then this algorithm would be the same as the well-known leapfrog algorithm. However, the split-operator approach provides a powerful framework capable of dealing with the far more complicated Hamiltonians often encountered in accelerator physics. The method is easily generalized to more terms (i.e., more splittings) if necessary. Furthermore, symplectic split-operator methods are easily generalized to higher order accuracy. If $\mathcal{M}_{2n}$ denotes a time-reversible approximation that is accurate to order $2n$, then the following is accurate to order $2n + 2$ [26],

$$\mathcal{M}(s)_{2n+2} = \mathcal{M}(x_0 s)_{2n}\,\mathcal{M}(x_1 s)_{2n}\,\mathcal{M}(x_0 s)_{2n}, \tag{14}$$

where x_0 and x_1 are given by

$$x_0 = \frac{1}{2 - 2^{1/(2n+1)}} \tag{15}$$

$$x_1 = \frac{-2^{1/(2n+1)}}{2 - 2^{1/(2n+1)}}. \tag{16}$$

There are also implicit symplectic methods based on this approach that do not require the Hamiltonian to be split into a sum of exactly solvable pieces [20].

If we treat $\mathcal{M}_1$ as corresponding to the external fields and $\mathcal{M}_2$ as corresponding to the space-charge fields, Eq. (9) describes an algorithm treating both single-particle magnetic optics effects and space-charge effects. A time step involves the following: (1) transport of a numerical distribution of particles through half a step based on $\mathcal{M}_{\text{ext}}$, (2) solution of Poisson's equation based on the particle positions and performance of a space-charge "kick" (i.e., an instantaneous change in momenta, since H_{sc} depends only on coordinates, hence $\mathcal{M}_{\text{sc}}$ only affects momenta), and (3) transport through the remaining half of the step based on $\mathcal{M}_{\text{ext}}$. If the space charge is intense, this can be performed repeatedly on successive pieces of a beamline element; if the space charge is weak, it may be possible to achieve good accuracy by computing the space-charge kicks infrequently, in which case $\mathcal{M}_{\text{ext}}$ would correspond to a string of elements within a half step. Thus, an important feature of this approach is that it enables one to use large time steps (i.e., large steps in the independent variable) in the regime of weak or moderate space charge. Essentially, it enables one to decouple the rapid variation of the externally applied fields from the more slowly varying space-charge fields. If more accuracy is required, one can use the fourth-order method of Forest and Ruth [21]; however, this requires three space-charge calculations per full step instead of just one, and since this dominates the execution time it is costly.

Finally, a subtle point is that, while most beam dynamics codes use a coordinate as the independent variable (typically the longitudinal coordinate z in a linac code), Poisson's equation has to be solved *at a fixed time*. Thus, prior to every space-charge calculation it is necessary to convert from the canonical coordinates and momenta currently in use to the more usual coordinates and momenta in which (x, y, z) are known at a fixed time. Such a calculation makes sense and is easily accomplished if the particle motion is essentially ballistic over a distance corresponding to the bunch length, since a bunch contains a distribution of arrival times, and they must all be moved to a fixed time.

In summary, split-operator methods provide the "glue" to join two major fields: magnetic optics and parallel particle simulation techniques. All that is required is (1) the ability to compute maps corresponding to external fields, (2) the ability to compute the space-charge fields (normally accomplished using a parallel Poisson solver), and (3) a knowledge of the particle positions at fixed time or the ability to compute them.

The physical system for beam dynamics studies consists of the beam and the accelerating/ transport system, which in turn contains a number of accelerating and focusing elements. In the present paper, these elements consist of drift spaces, magnetic quadrupoles, and radio-frequency cavities (RF accelerating gaps). Let K_{ext} denote the Hamiltonians corresponding to these elements, with the axial coordinate z as the independent variable. In this case, the canonical coordinates and momenta are (X, P_x, Y, P_y, T, P_t), where T denotes arrival time and P_t denotes the negative energy. The Hamiltonians for various beamline elements are as follows:

For the drift tube, K_{ext} is [27]

$$K_{\text{ext}} = \frac{1}{2}\left(P_x^2 + P_y^2\right) + \frac{1}{2\gamma_0^2\beta_0^2}P_t^2. \tag{17}$$

For the magnetic quadrupole, K_{ext} is [27]

$$K_{\text{ext}} = \frac{1}{2}\left(P_x^2 + P_y^2\right) + \frac{1}{2}k(z)(X^2 - Y^2) + \frac{1}{2\gamma_0^2\beta_0^2}P_t^2, \tag{18}$$

where the focusing strength, $k(z)$, is related to the quadrupole gradient according to

$$k(z) = \frac{q}{p^0}g(z). \tag{19}$$

For the accelerating RF gap, K_{ext} is [28]

$$\begin{aligned} K_{\text{ext}} = {} & \frac{\delta}{2lp^0}\left(P_x^2 + P_y^2\right) + \frac{l}{2\delta}\left[\frac{1}{p^0}\left(\frac{q}{2\omega}e'\sin\phi_s\right)^2 - \frac{q}{2\omega}\left(e'' + \frac{w^2}{c^2}e\right)\sin\phi_s\right](X^2 + Y^2) \\ & - \frac{qe'\sin\phi_s}{2p^0\omega}(XP_x + YP_y) + \frac{m^2\omega^2 l\delta}{2(p^0)^3}P_t^2 - \frac{\omega qe\sin\phi_s}{2\omega^2 l\delta}T^2, \end{aligned} \tag{20}$$

where e is the on-axis electric field in the gap, ϕ_s is given by $\phi_s = \omega t^g(z) + \theta$, and $p^0(z)$ is the design momentum. In order to compute $\mathcal{M}_{\text{ext}}$ for the RF gap, one needs to numerically solve the equations of motion for the design trajectory inside the gap. These equations are given by

$$(t^{\text{g}})' = \frac{-p_t^g/c}{\sqrt{\left(p_t^g\right)^2 - m^2c^4}} \tag{21}$$

$$\left(p_t^{\text{g}}\right)' = -qe(z)\cos(\omega t^g + \theta). \tag{22}$$

Last, in addition to the Hamiltonians for the various external elements, we need the Hamiltonian corresponding to the space-charge field. This is given by

$$K_{\text{self}} = \frac{q/\delta c}{l\beta^{\text{g}}(\gamma^g)^2}\phi, \tag{23}$$

which includes the effect of both the electrostatic field and the azimuthal magnetic field. The potential ϕ can be obtained by convolving the charge density with a Green's function. In our code, the charge density is obtained by depositing the particles onto a grid using a cloud-in-cell (CIC) scheme. The same scheme is used to interpolate the field from the grid back onto the particles. The potential is expressed as

$$\phi_{p,q,r} = \sum G_{p-p',q-q',r-r'}\rho_{p',q',r'}, \tag{24}$$

where G is the Green's function on the grid, and ρ is the charge density on the grid. Often, the beam size is much smaller than the inside wall radius of the accelerator, in which case we may treat the beam as an isolated system. In such a case, the above convolution can be calculated using a Fast fourier transform (FFT) technique given by Hockney and Eastwood [29].

III. PARALLEL PARTICLE-IN-CELL ALGORITHM

A message-passing programming paradigm with MPI (message-passing interface) is employed in our parallel PIC simulation. MPI is a standard library of message-passing programs bound to C (C++) and Fortran [30]. In this paradigm, a computer program creates one or more processes. Each process can execute the same program or a different program with local data. In most implementations, each process is mapped to a physical processor with a unique identification number. When the data from more than one processor are required, explicit communication is performed by calling library routines to send or receive messages from other processors. Hence, in this programming model, the programmer has to control the data distribution on the processors and communication among processors. This gives it the advantage of flexibility compared with the data-parallel programming model. However, this also increases the difficulty of parallel programming compared with the data-parallel programming model. Applying object-oriented design to parallel message-passing programming helps to encapsulate the details of communication and data distribution. This enables the user to manage the applications at a higher level.

A two-dimensional domain-decomposition approach is employed in our parallel particle simulation [10, 14]. A schematic plot of the two-dimensional decomposition on the *y*–*z* plane is shown in Fig. 1. The solid grid lines define the computational domain grids. The dashed lines define the local computational domain on each processor. Here, the boundary grids are the outermost grids inside the physical boundary. The guard grids are used as temporary storage of grid quantities from the neighboring processors. The physical computational domain is defined as a three-dimensional rectangular box with range $x_{\min} \leq x \leq x_{\max}$, $y_{\min} \leq y \leq y_{\max}$, and $z_{\min} \leq z \leq z_{\max}$. This domain is decomposed on the *y*–*z* plane into a number of small rectangular blocks. These blocks are mapped to a logical two-dimensional Cartesian processor grid. Each processor contains one rectangular block domain. The range

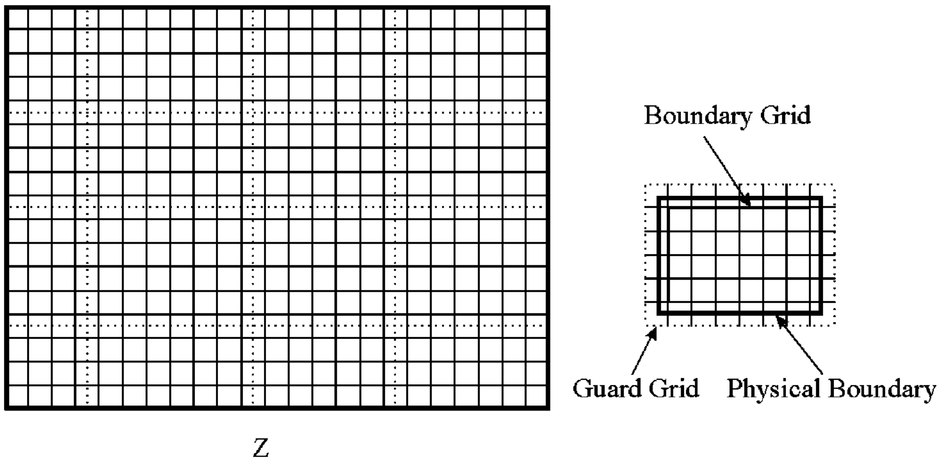

FIG. 1. A schematic plot of two-dimensional decomposition on the *y*–*z* domain.

of a block on a single processor is defined as $x_{\text{min}} \leq x \leq x_{\text{max}}$, $y_{\text{lcmin}} \leq y \leq y_{\text{lcmax}}$, and $z_{\text{lcmin}} \leq z \leq z_{\text{lcmax}}$. Here, the subscripts lcmin and lcmax specify local minimum and local maximum. The mesh grid is defined to store the field-related quantities such as charge density and electric field. The number of grid points along three dimensions on a single processor is defined as

$$Nx_{\text{local}} = \text{int}[(x_{\text{max}} - x_{\text{min}})/hx] + 1 \tag{25}$$

$$Ny_{\text{local}} = \text{int}[(y_{\text{lcmax}} - y_{\text{min}})/hy] - \text{int}[(y_{\text{lcmin}} - y_{\text{min}})/hy] + N_{\text{g}} \tag{26}$$

$$Nz_{\text{local}} = \text{int}[(z_{\text{lcmax}} - z_{\text{min}})/hz] - \text{int}[(z_{\text{lcmin}} - z_{\text{min}})/hz] + N_{\text{g}}, \tag{27}$$

where hx, hy, and hz are the mesh sizes along the x, y, and z directions, respectively. The quantity N_{g} refers to the number of guard grids in Ny_{local} and Nz_{local}. $N_{\text{g}} = 2$ if the number of processors in that dimension is greater than 1; otherwise, $N_{\text{g}} = 1$. For the processor containing the starting grid in the global mesh, there is one more grid point along the y and z directions. The particles with spatial positions within the local computational boundary are assigned to the processor containing that part of physical domain.

The parallel computation starts with constructing a 2-D logical Cartesian processor grid, reading input data from processor 0 and broadcasting it to the other processors, setting up the local initial computational domain, initializing objects, and generating particles from the initial distribution function. There are three approaches to generating the particles local to the processor at the beginning of the simulation. In the first approach, each processor generates the average number of particles by sampling the whole initial distribution. Then explicit all-to-all communication is used to send the particles to the appropriate processors. This approach has the advantage that each processor need only generate a fraction of the particles. However, the communication cost will increase with increasing numbers of processors and particles, which makes this approach less scalable. In the second method, each processor generates all the particles to be used in the simulation; only particles local to the processor are kept, and the other particles not local to the computational domain are thrown away. This approach avoids the communication cost and can be used when the problem size and the number of processors used are not large, e.g., in our performance benchmark simulations where the initialization time is negligible. Nevertheless, this approach is extremely inefficient when a large number of processors are used since the time cost is the same regardless of the number of processors. In the third approach, each processor generates the average number of particles by sampling a part of the initial distribution using a rejection method [31]. This part of the initial distribution contains the computational domain local to the processor. Hence, the particles generated from this distribution will be local to the processor. There is no need for communication. This approach has the advantage of scaling with increasing number of processors. The disadvantage of this approach is that the result may not be reproducible using a different number of processors partly because of the difference in random number generation on each processor. This approach was used to generate an initial Gaussian distribution for the simulation of the APT superconducting linac described in Section VI.

The particles generated on each processor advance following the maps defined in Section II. If a particle moves outside the local computational domain, it is sent to the processor corresponding to where it is located. A particle manager function is defined to handle explicit communication among two-dimensional processor grids using MPI. The y and z positions of every particle on each processor are checked. The particle is copied to one of its four buffers and sent to one of its four neighboring processors when its y or z position is outside the local computational domain. After a processor receives the particles from its neighboring processors, it will decide whether to send some of them further out. The outgoing particles are counted and copied into four temporary arrays. The remaining particles are copied into another temporary array. This process is repeated until no outgoing particle is found on any processor. Then, the particles in the temporary storage, along with the particles left in the original particle array, are copied into a new particle array.

After each particle moves to its local computational domain, a linear CIC particle-deposition scheme is carried out for all processors to determine the charge density on the grid. The particles located between the boundary grid and the computational domain boundary will also contribute to the charge densities on the boundary grids of neighboring processors. Hence, explicit communication is required to send the charge density on the guard grids, which is from local particle deposition, to the boundary grids of neighboring processors to sum up the total charge density on the boundary grids. With the charge density on the grids, Hockney's FFT algorithm is used to solve Poisson's equation with open boundary conditions. Due to this algorithm, the original grid number is doubled in each dimension. The charge density on the original grid is kept the same, and the charge density elsewhere is set to 0. The Green's function on the original grid is defined as

$$G_{p,q,r} = \frac{1}{\sqrt{(hx(p-1))^2 + (hy(q-1))^2 + (hz(r-1))^2}}, \tag{28}$$

where $p = 1, \ldots, Nx + 1$, $q = 1, \ldots, Ny + 1$, and $r = 1, \ldots, Nz + 1$. Here, Nx, Ny, and Nz are the computation grid number, in all three dimensions. For points outside the original grid, symmetry is used to define Green's function according to

$$G_{p,q,r} = G_{2Nx-p+2,q,r} \tag{29}$$

$$G_{p,q,r} = G_{p,2Ny-q+2,r} \tag{30}$$

$$G_{p,q,r} = G_{p,q,2Nz-r+2}, \tag{31}$$

where $p = Nx^*_{\text{local}} + 2, \ldots, 2Nx$, $q = Ny^*_{\text{local}} + 2, \ldots, 2Ny$, and $r = Nz^*_{\text{local}} + 2, \ldots, 2Nz$. Communication is required to double the original distributed three-dimensional grid explicitly. This can be avoided by including this process in the three-dimensional FFT. In the three-dimensional parallel FFT, we have taken advantage of the undistributed dimension along the x dimension, where a local serial FFT can be done in that dimension for all processors. A local temporary two-dimensional array with size $(2Nx, Ny_{\text{local}})$ is defined to contain part of the charge density at fixed z. The charge density on the original grid is copied into the (Nx, Ny_{local}) part of the temporary array. The rest of the temporary array is filled with 0. In regard to the FFT of the Green's function, symmetry can be used to obtain the values of the Green's function in the region $(Nx + 2, Ny_{\text{local}})$. After the local two-dimensional FFT along x is done, it is copied back to a slice of a new three-dimensional array with size $(2Nx, Ny_{\text{local}}, Nz_{\text{local}})$. A loop through Nz_{local} gives the FFT along x for the three-dimensional array. Then, a transpose is used to switch the x and y indices. Now, the three-dimensional matrix has size $(Ny, Nx'_{\text{local}}, Nz_{\text{local}})$. Here, Nx'_{local} is the new local number of grids in the x dimension along the y dimension processors. A similar process is performed to obtain the FFT along the y direction for a double-size grid of size $(2Ny, Nx'_{\text{local}}, Nz_{\text{local}})$. Another transpose is used to switch the y and z indices and a local FFT along z with a double-size grid is done on all processors to finish the three-dimensional FFT for the double-size grid in all three dimensions. During the inverse parallel FFT, a reverse process is employed to obtain the potential on the original grids. In the transpose of indices, global all-to-all communication is used.

From the potential on the grid, we calculate the electric field on the grid using a central finite difference scheme. To calculate the electric field on a boundary grid, the potential on a boundary grid of neighboring processors is required. A communication pattern similar to that employed in the charge density summation on the boundary grids is used to send the potential from the boundary grids to the guard grids of neighboring processors. After the electric field on the grids is obtained, it has to be interpolated from the grids onto the local particles to push the particles. Since we have used the linear CIC scheme, the electric field of the particles between the boundary grid and the computational domain boundary will also depend on the electric fields on the boundary grids of neighboring processors. A similar communication pattern is used to send the electric field from the boundary grids to the guard grids of the neighboring processors. With the electric fields on grids local to each processor, the interpolation is done for all processors to obtain the space-charge force on every particle. The local particles are updated in momentum space based on the space-charge force. This operation defines the mapping $\mathcal{M}_2$.

Dynamic load balancing is employed with adjustable frequency to keep the number of particles on each processor approximately equal. A density function is defined to find the local computational domain boundary so that the number of particles on each processor is roughly balanced. This number depends on the local integration of the charge density on each processor. To determine the local boundary, first the three-dimensional charge density is summed up along the x direction on each processor to obtain a two-dimensional density function. This function is distributed locally among all processors. Then, the two-dimensional density function is summed up along the y direction to get the local one-dimensional charge density function along z. This density function is broadcast to the processors along the y direction. The local charge density function is gathered along z and broadcast to processors along the z direction to get a global z direction charge density distribution function on each processor. Using this global z direction density distribution, the local computational boundary in the z dimension can be determined assuming that each processor contains a fraction of the total number of particles about equal to $1/\text{nproc}_z$. Here, nproc_z is the number of processors along the z direction in the two-dimensional Cartesian processor grid. A similar process is used to determine the local computational boundary in the y direction. Strictly speaking, the above algorithm will work correctly for a two-dimensional density distribution function which can be separated as a product of two one-dimensional functions along each direction. However, from our experience, this algorithm works reasonably well for the distributions generally produced in beam dynamics simulations in a linear accelerator.

IV. OBJECT-ORIENTED SOFTWARE DESIGN

The above parallel particle-in-cell algorithm is implemented in an object-oriented framework for accelerator simulation. Object-oriented software design is a method of design encompassing the process of object-oriented decomposition [32]. The complex physical system is analyzed and then decomposed into simpler physical modules. Next, objects are identified inside each module, and classes are abstracted from these objects. Each class has interfaces to communicate with the outside environment. Relationships are then built up among different classes and objects. These classes and objects are implemented in a concrete language representation. The implemented classes and objects are tested separately and then put into the physical module. Each module is tested separately before it is assembled into the whole program. Finally, the whole program is tested to meet the requirements of the problem.

Our application of this object-oriented design methodology to beam dynamics studies in accelerators results in the decomposition of the physical system into five parts. The first part handles the particle information and consists of the *Beam*, *BeamBC* (i.e, beam boundary condition), and *Distribution* classes. The second part handles information regarding quantities defined on the field grid and consists of the *Field* and *FieldBC* classes. The third part handles the external focusing and accelerating elements and consists of the *BeamLineElem* base class and its derived classes, the *Drift Tube* class, the *Quadrupole* classes, and the *RF gap* class. The fourth part handles the computational domain geometry and consists of the *Geometry* class. The last part provides auxiliary and low-level classes to handle explicit communication and input–output containing the *Pgrid2d*, *Communication*, *Utility*, *InOut*, and *Timer* classes. The class diagram of the object-oriented model for a beam dynamics system is presented in Fig. 2. Here, run-time polymorphism is used to implement different external beamline elements. A single operation using the function of the beamline-element base class can automatically select the appropriate function from different concrete beamline-element class objects to execute. The *Inheritance* relation in Fig. 2 defines an “is” relationship among classes. The *Aggregation* defines the relation that a class has an object of another class as one of its data members. The *Use* defines a relation that a class uses an object of another class in one or more member functions. The above object-oriented design is implemented using Fortran 90 (F90). Even though F90 does not provide explicit language support for some object-oriented programming features like inheritance and polymorphism, these features can be emulated using user-defined data types, pointers, and modules in F90. The inheritance of data members can be implemented by including exactly one instance of the base class data member in a derived class. The inheritance of methods can be implemented

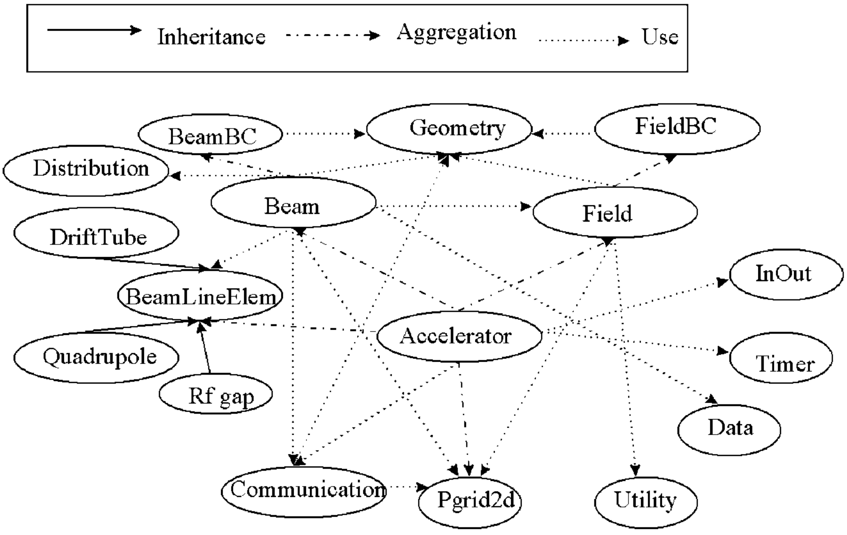

FIG. 2. Class diagram of accelerator beam dynamics system.

by delegating to the base class the responsibility of carrying out the operation on the base class component of the derived class object. The polymorphism can be implemented by constructing a pointer object that can point to any member in an inheritance hierarchy and a dispatch mechanism that can select the appropriate procedure to execute based on the actual class referenced in the pointer object. More detailed discussions about expressing object-oriented concepts in F90 can be found in Refs. [33, 34]. Some important class interfaces of accelerator beam dynamics systems have been discussed in another publication [35]. A similar code was also developed using the POOMA C++ framework [36]. In this paper, we only show the simulation results using the F90/MPI code.

V. PERFORMANCE TEST

The performance of the object-oriented code was tested on both the SGI/Cray T3E-900 and the SGI Origin 2000. The SGI/Cray T3E-900 is a scalable, logically shared, physically distributed multiprocessor machine with a range of configurations up to thousands of processors [37]. Each node consists of a DEC Alpha 64-bit RISC microprocessor, local memory, system control chip, and some network interfaces. The RISC microprocessor is cache-based, has pipelined functional units, and issues multiple instructions per cycle. The clock speed is 450 MHz. Each node has its own local DRAM memory with a capacity of from 64 megabytes (MB) to 2 gigabytes (GB). A shared, high-performance, globally addressable memory subsystem makes these memories accessible to every node. There are two-level on-chip caches which can only be cached by local memory: one with 8 KB instruction and data caches and another with 96 KB three-way associative cache. The nodes are connected by a high-bandwidth, low-latency bidirectional 3-D torus interconnect network system.

The SGI Origin 2000 is a scalable, distributed shared-memory multiprocessor machine. It consists of a number of processing nodes linked together by a multidimensional interconnection fabric. Each processing node contains either one or two processors and a portion of shared memory, which is physically distributed locally to each node but is also accessible to all other processors through the interconnection fabric. Each node also contains a directory for cache coherence and two interfaces to connect to I/O devices and to link system nodes through the interconnection fabric. The processor used in the SGI Origin 2000 is the MIPS R10000, a high-performance 64-bit superscalar processor with up to 4 GB memory, 32 KB on-chip data cache, 32 KB on-chip instruction cache, and 4 MB secondary cache. The single-node clock speed for the system we used is 250 MHz. A cabinet can consist of up to 128 processor nodes [38].

The effect of dynamic load balancing is exhibited in Fig. 3. It shows the largest and smallest number of particles on one processor, as a function of time, with and without dynamic load balancing, on 16 processors. The total number of simulated particles is 2 million with a $64 \times 64 \times 64$ grid. We see here that with dynamic load balancing the difference between the maximum number of particles and the minimum number of particles has been drastically reduced. This demonstrates that the dynamic load balancing algorithm described above works well. Here, we have done dynamic load balancing every time step. The overhead associated with the dynamic load balancing operations is about 10% to 20% of the total time. This depends on the dynamics of the physical problem and on the shape of the resulting beam density and its distribution among processors. Typically, we call one dynamic load balancing every five steps.

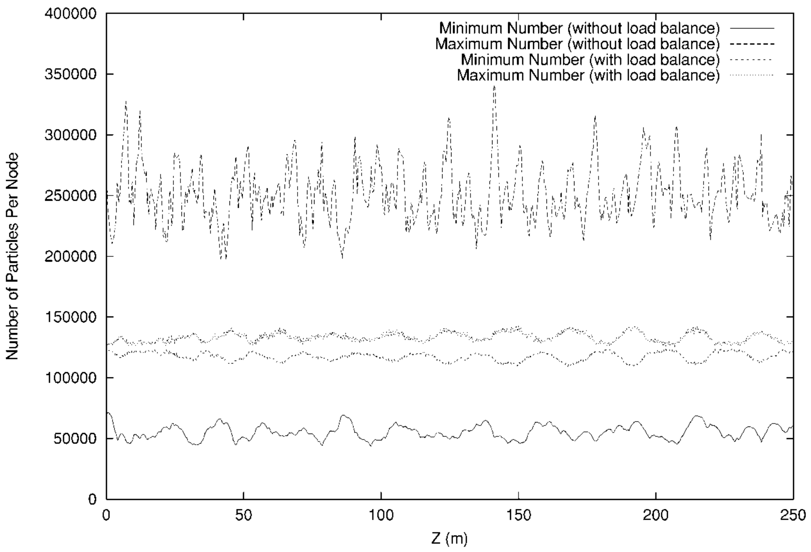

FIG. 3. The maximum and minimum numbers of particles on one processor without and with dynamic load balance.

In Fig. 4, we give a comparison of the execution time on the Cray T3E as a function of the number of processors using one-dimensional and two-dimensional parallel processor partitions. In this simulation, we have used 2.6 million particles and a $64 \times 64 \times 64$ grid. The two-dimensional partition shows better scalability and is faster than the one-dimensional partition because a two-dimensional processor partition has a more favorable surface-to-volume ratio. Communication cost is proportional to the surface area of the subdomain, whereas computation is proportional to its volume.

Fig. 5 shows the execution time as a function of the processor number on the SGI/Cray T3E-900 and on the SGI Origin 2000 for the same problem as in Fig. 4. Good scalability

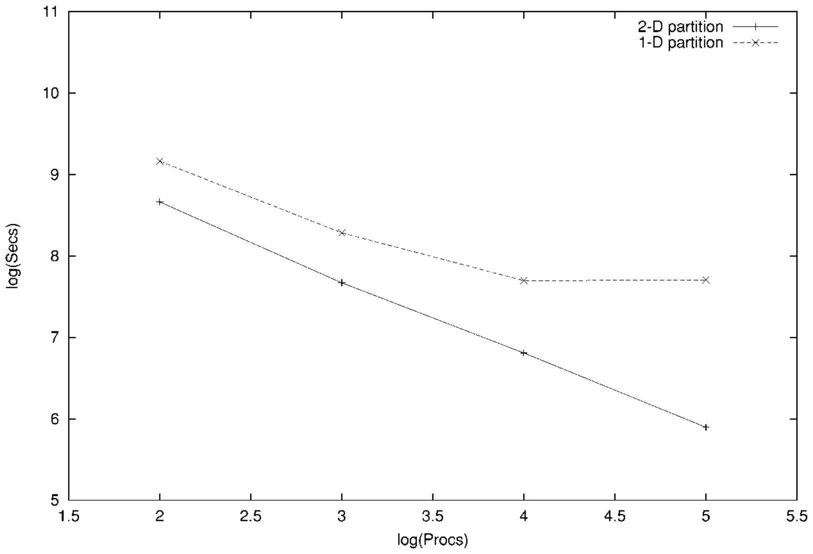

FIG. 4. The time cost as a function of the number of processors on the Cray T3E using one-dimensional and two-dimensional parallel partitions.

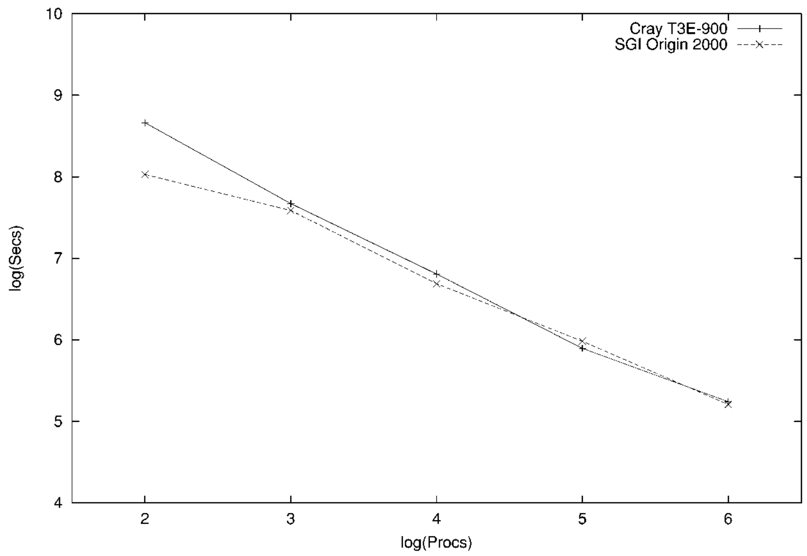

FIG. 5. The time cost as a function of the number of processors on the Cray T3E and the SGI Origin 2000.

of our object-oriented parallel PIC code has been achieved. The general performance of our code on the Origin and T3E machines beyond four processors is nearly identical. This coincidence may be due to the slower CPU speed but larger cache size of the Origin machine than that of the T3E. Another factor that could contribute to this is that the code runs as single precision on the SGI machine but will run as double precision on the T3E machine by default.

To see how well this code will scale for larger problem sizes, Fig. 6 shows, for the SGI machine, the speedup normalized by the time on four processors as a function of the number of processors, with three different problem sizes. Here, the number of simulation particles

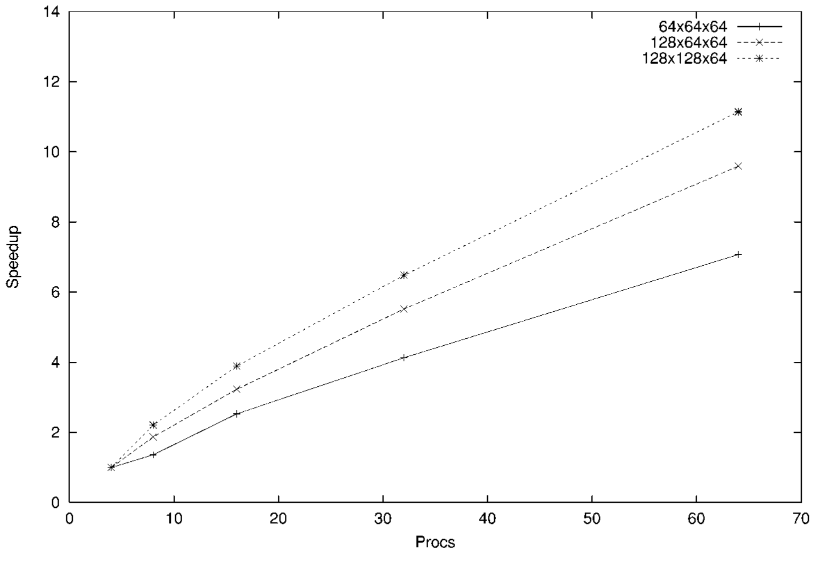

FIG. 6. The speedup as a function of the number of processors on the SGI Origin 2000 with different problem sizes.

TABLE I
Physical Parameters in the APT Design

Energy gain	211.4–1.03 GeV
Beam current	0.1 A
Accelerator length	513.58 m (includes three sections)
Quadrupole gradient	5.60–5.10, 5.50–6.05, 5.00–7.25 T/m
Accelerating gradient	4.30–4.54, 4.30–5.01, 5.246 MV/m
Synchronous phase	-30° to -35°, -30° to -42°, -30°

used per grid point is fixed as 10 for all three cases. It is seen that with increasing problem size, the speedup also increases. This example shows that the code is more efficient with larger problem sizes.

VI. APPLICATION

As an application, we simulated beam transport through three superconducting sections in a design of the APT linac [39]. The first section accelerates the beam from 211.4 to 242.0 MeV and contains six 2-cavity cryomodules. The second section accelerates the beam from 242.0 to 471.40 MeV and contains thirty 3-cavity cryomodules. The third section accelerates the beam to 1.03 GeV and contains thirty-five 4-cavity cryomodules. The major physical parameters in the design are listed in Table 1.

The external focusing and accelerating fields for the first two cryomodules are given in Fig. 7. A quadrupole doublet focusing lattice is used to provide transverse strong focusing and to reduce the focusing period compared with that for a singlet lattice. The external longitudinal RF field is obtained from a MAFIA [40] calculation of the 5-cell superconducting cavity. For the above physical parameters and external field, we performed the simulation using 20 million simulation particles on a $128 \times 128 \times 128$ grid. The initial distribution

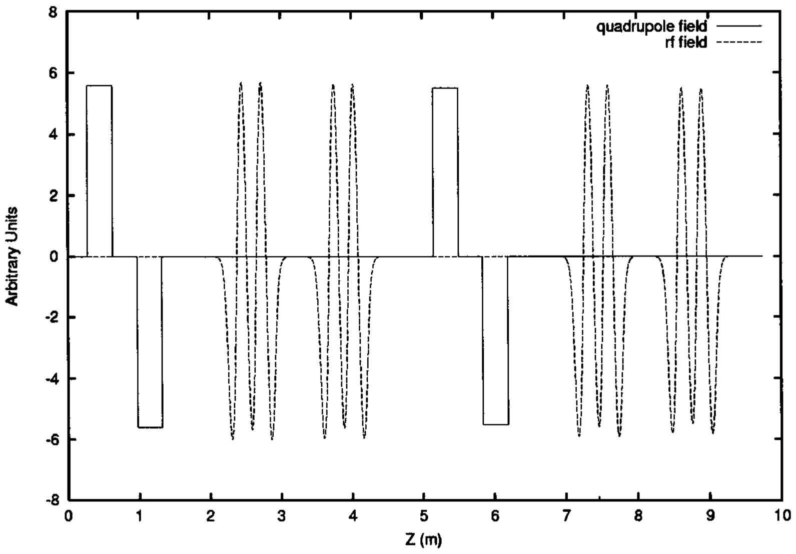

FIG. 7. The external focusing and accelerating field in the superconducting linac.

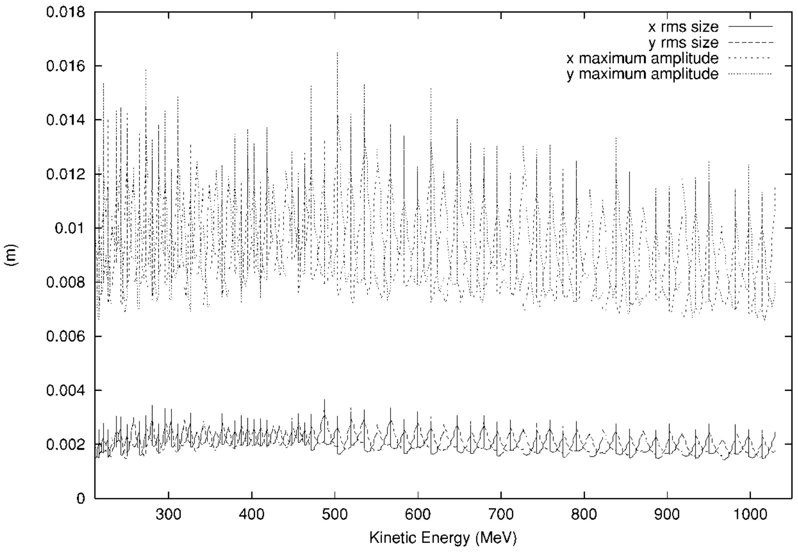

FIG. 8. The transverse rms size and maximum amplitudes of the beam as functions of its kinetic energy.

used here is a six-dimensional Gaussian distribution in phase space. Figure 8 gives the transverse rms beam size and maximum amplitudes as a function of kinetic energy of the beam. A jump in transverse rms beam size around 480 MeV is due to the jump in external focusing between the second section and the third section. The maximum transverse amplitudes determine the minimum aperture that can be used without particles striking the beam pipe. Figure 9 shows the longitudinal phase space plot at the end of the linac. The spiral structure suggests the formation of a beam halo due to the mismatched focusing which can be understood using a particle-core model [3]. Particles in the beam halo will be lost if they move to large amplitude and strike the beam pipe. The resulting radioactivity is a major

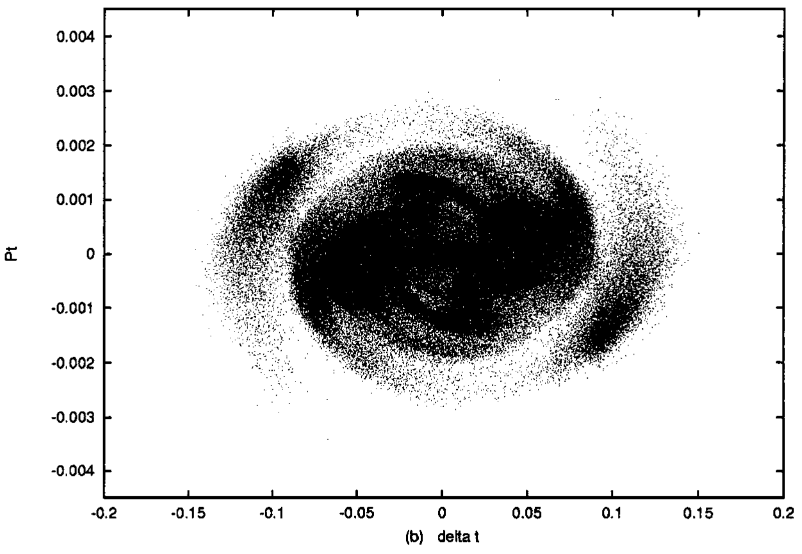

FIG. 9. The longitudinal phase space plots at the end of the linac.

issue for high current machines because it affects the safety, reliability, and availability of the accelerator.

VII. CONCLUSIONS

In this paper, we have presented an object-oriented three-dimensional parallel particle-in-cell program for beam dynamics simulation in linear accelerators. This program employs a domain decomposition method with MPI. A dynamic load balance scheme is implemented in the code. The use of object-oriented techniques results in better maintainability, reusability, and extensibility compared with a conventional structure-based approach. Performance tests on the SGI/Cray T3E-900 and the SGI Origin 2000 showed good scalability. This code was successfully applied to the simulation of beam transport through three superconducting sections in the APT linac.

ACKNOWLEDGMENTS

We thank Drs. Barbara Blind, Frank Krawczyk, and Thomas Wangler for the RF superconducting data. This work was performed on the SGI/Cray T3E at the National Energy Research Scientific Computing Center located at Lawrence Berkeley National Laboratory and the SGI Origin 2000 at the Advanced Computing Laboratory, Los Alamos National Laboratory. This work was supported by the DOE Grand Challenge in Computational Accelerator Physics and by the Los Alamos Accelerator Code Group.

REFERENCES

1. F. Sacherer, *IEEE Trans. Nucl. Sci.* **NS-18**, 1105 (1971).
2. J. Struckmeier and M. Reiser, *Part. Accel.* **14**, 227 (1984).
3. R. L. Gluckstern, *Phys. Rev. Lett.* **73**, 1247 (1994).
4. R. Ryne and S. Habib, in *Computational Accelerator Physics*, edited by J. J. Bisognano and A. A. Mondelli, AIP Conference Proceedings 391 (Woodbury, New York, 1997) p. 377.
5. B. B. Godfrey, in *Computer Applications in Plasma Science and Engineering*, edited by A. T. Drobot (Springer–Verlag, New York, 1991).
6. M. E. Jones, B. E. Carlsten, M. J. Schmitt, C. A. Aldrich, and E. L. Lindman, *Nucl. Instrum. Methods Phys. Res. A* **318**, 323 (1992).
7. A. Friedman, D. P. Grote, and I. Haber, *Phys. Fluids B* **4**, 2203 (1992).
8. R. Ryne, ed., *Computational Accelerator Physics*, AIP Conference Proceedings 297 (Los Alamos, NM, 1993).
9. H. Takeda and J. H. Billen, Recent developments of the accelerator design code PARMILA, in *Proc. XIX International Linac Conference, Chicago, August 1998*, p. 156.
10. P. C. Liewer and V. K. Decyk, *J. Comput. Phys.* **85**, 302 (1989).
11. C. S. Lin, A. L. Thring, J. Koga, and E. J. Seiler, *J. Parallel Distributed Comput.* **8**, 196 (1990).
12. R. Ferrell and E. Bertschinger, *Int. J. Mod. Phys. C* **5**, 933 (1994).
13. V. K. Decyk, *Comput. Phys. Commun.* **87**, 87 (1995).
14. P. M. Lyster, P. C. Liewer, R. D. Ferraro, and V. K. Decyk, *Comput. Phys.* **9**, 420 (1995).
15. J. V. W. Reynders and D. W. Forslund *et. al., Comput. Phys. Commun.* **87**, 212 (1995).
16. J. Wang, P. Liewer, and V. Decyk, *Comput. Phys. Commun.* **87**, 35 (1995).
17. H. X. Vu, *J. Comput. Phys.* **144**, 257 (1998).
18. R. Ryne, S. Habib, J. Qiang, K. Ko, Z. Li, B. McCandless, W. Mi, C. Ng, M. Saparov, V. Srinvas, Y. Sun, X. Zhan, V. Decyk, and G. Golub, The U.S. DOE Grand Challenge in Computational Accelerator Physics, in *Proc. XIX International Linac Conference, Chicago, IL, August 1998*, p. 701.

19. A. J. Dragt, *Part. Accel.* **55**, 499 (1996).

20. E. Forest *et al., Phys. Lett. A* **158**, 99 (1991). [Note that there is a typographical error in Eq. 2.5, which is the main result due to Yoshida. The quantities z_0 and z_1 are interchanged.]

21. E. Forest and R. Ruth, *Physica D* **43**, 105 (1990).

22. P. J. Channell and F. R. Neri, *Fields Inst. Commun.* **10**, 45 (1996).

23. J. E. Campbell, *Proc. London Math. Soc.* **29**, 14 (1898).

24. H. F. Baker, *Proc. London Math. Soc.* **34**, 347 (1902).

25. F. Hausdorff, *Math. Naturwiss.* **58**, 19 (1906).

26. H. Yoshida, *Phys. Lett. A* **150**, 262 (1990).

27. R. D. Ryne, *Computational Methods in Accelerator Physics*, in preparation, 1999.

28. R. D. Ryne, *The Linear Map for an rf Gap Including Acceleration*, LANL Report 836 R5 ST 2629 (1991).

29. R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles* (Hilger, New York, 1988).

30. M. Snir, S. Otto, S. H. Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference, Vol. 1* (MIT Press, Cambridge, MA, 1998).

31. M. H. Kalos and P. A. Whitlock, *Monte Carlo Methods* (Wiley, New York, 1986).

32. G. Booch, *Object-Oriented Analysis and Design with Applications* (Benjamin–Cummings, Menlo Park, CA, 1994).

33. V. K. Decyk, C. D. Norton, and B. K. Szymanski, *Sci. Programming* **6**, 363 (1997).

34. V. K. Decyk, C. D. Norton, and B. K. Szymanski, *Comput. Phys. Commun.* **115**, 9 (1998).

35. J. Qiang, R. D. Ryne, and S. Habib, Fortran implementation of object-oriented design in parallel beam dynamics simulations, *Comput. Phys. Commun.*, to appear.

36. W. Humphrey, R. Ryne, and T. Cleland *et al.*, in *Computing in Object-Oriented Parallel Environments*, edited by D. Caromel, R. R. Oldehoeft, and M. Tholburn, Lecture Notes in Computer Science, (Springer–verlag, Berlin, New York, 1998), Vol. 1505.

37. http://www.cray.com/products/systems/cray3e/overview.html (1998).

38. http://www.sgi.com/origin/2000/(1998).

39. G. P. Lawrence, High-power proton Linac for APT: Status of design and development, in *Proc. Linac98, Chicago, IL, 1998*.

40. T. Weiland, Int. J. Numerical Model. **9**, 295 (1996).