Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Computational Physics 205 (2005) 755–775
JOURNAL OF COMPUTATIONAL PHYSICS
www.elsevier.com/locate/jcp

A new asynchronous methodology for modeling of physical systems: breaking the curse of courant condition

H. Karimabadi *, J. Driscoll, Y.A. Omelchenko, N. Omidi

Scibernet, Inc., Computational Physics, 12837 Caminito del Canto, San Diego, CA 92014, USA

Received 7 June 2004; received in revised form 14 October 2004; accepted 1 December 2004
Available online 22 January 2005

Abstract

Computer simulation of many important complex physical systems has reached a plateau because most conventional techniques are ill equipped to deal with the multi-scale nature of such systems. The traditional technique to simulate physical systems modeled by partial differential equations consists of breaking the simulation domain into a spatial grid and then advancing the state of the system synchronously at regular discrete time intervals. This so-called time-driven (or time-stepped) simulation (TDS) has inherent inefficiencies such as the time step restriction imposed by a global CFL (Courant–Friedrichs–Levy) condition. There is ongoing research on introducing local time refinement (local versus global CFL) within the time-stepped methodology. Here, we propose an entirely different (asynchronous) simulation methodology which uses the spatial grid but the time advance is based on a discrete event-driven (as opposed to time-driven) approach. This new technique immediately offers several major advantages over TDS. First, it allows each part of the simulation, that is the individual cells in case of fluid simulations and individual particles within a cell in case of particle simulations, to evolve based on their own *physically* determined time scales. Second, unlike in the TDS where the system is updated faithfully in time based on a pre-selected user specified time step, here the role of time step is replaced by a threshold for local state change. In our technique, individual parts of the global simulation state are updated on a "need-to-be-done-only" basis. Individual parts of the simulation domain set their own time scales for change, which may vary in space as well as during the run. In a particle-in-cell (PIC) simulation, DES enables a self-adjusting temporal mesh for each simulation entity down to assigning an individual time step to each particle. In this paper, we illustrate this new technique via the example of a spacecraft charging in a neutral plasma due to injection of a charged beam particle from its surface and compare its performance with the traditional techniques. We find that even in one-dimension, the new DES technology can be more than 300 times faster than the traditional TDS. Aside from sheer performance advantages, the real power of this technique is in its inherent ability to adapt to the spatial inhomogeneity of the problem. This enables building intelligent algorithms where interaction of simulation entities (e.g., cells, particles) follow elementary rules set by the

* Corresponding author.
E-mail address: homa@ece.ucsd.edu (H. Karimabadi).

0021-9991/$ - see front matter
doi:10.1016/j.jcp.2004.12.003

underlying physics laws. Finally, our extensions of this technique to other problems such as the solution of diffusion equation and electromagnetic codes are briefly discussed.

1. Introduction

The traditional time advance of spatially discretized physical systems is based on the time-stepped (or time-driven) approach [1]. In its simplest form, the method involves updating discrete physical quantities on a uniform spatial mesh at regular (spatially uniform) time intervals derived from an explicit time discretization of the underlying model equations. Such methods require that the time step satisfy a Courant–Friedrichs–Levy (CFL) condition $\Delta t < \Delta x/V$, where V is the fastest physical speed in the problem and Δx is the mesh spacing. Simulation codes using a uniform mesh are quite prevalent even today and have been adequate for modeling simple systems. On the other hand, with gains in computer speed there has been a growing interest in modeling complex systems. Such systems are often highly inhomogeneous and may even require different physical/numerical models in different parts (regions) of the simulation domain. To address multi-scale, multi-physics systems new techniques have had to be devised. Much of the work to date has concerned the development of sophisticated spatial mesh techniques where local spatial refinement is introduced to resolve certain features in more detail. However, this local refinement reduces the allowable time step and can significantly degrade the performance advantage a non-uniform mesh provides. Therefore, much effort has gone into developing schemes within the time-stepped methodology to allow the time step to vary spatially so as to satisfy a local, rather than a global, CFL condition (e.g. [2] and references therein). In case of particle-in-cell (PIC) simulations, one is faced with the additional problem that individual particles evolve on different time scales than the fields. In order to decouple the evolution of particles and fields, Friedman et al. [3] proposed an implicit technique where grid quantities are interpolated in time to obtain the source contributions from groups of particles not advanced during the current step. The particles are put into different groups and advanced based on some multiple of the smallest time step. Aside from issues related to implicit nature of this technique (e.g., damped equations of motion that have to be used rather than the time reversible equations in explicit codes, extra complexity of the implicit particle and field calculations, etc.), fields are still updated based on the traditional TDS.

The most developed of the TDS techniques is the so-called adaptive mesh refinement (AMR) where local time stepping is done by automatically taking smaller time steps in regions where the mesh is refined. Examples of computational software based on the AMR technology include Chombo (developed at Lawrence Berkeley National Lab) and SAMRAI (developed at Lawrence Livermore National Lab). The AMR mesh is a hierarchy of nested meshes of varying spatial and temporal resolutions. Computational cells on each level of refinement are clustered into logically rectangular mesh regions, called patches. The time step taken in each patch is related to the patch cell size via a Courant condition local to the patch. Therefore, one has different temporal resolutions in different patches.

All traditional methods face the following common problems:

- Excessive computation – even in AMR codes the time step is still uniform within each patch and cell quantities are unnecessarily updated regardless of how fast they are changing in time.
- CFL constraint – the time step is constrained by the global CFL condition at the worst (in non-uniform mesh schemes) and local (to the patch) CFL at the best (in AMR schemes).

In short, the time-stepped methodology may not be an ideal foundation for development of multi-scale, multi-physics simulation codes. This is because the basic premise of the technique, system update at

spatially uniform time intervals, is at odds with the need to introduce a temporal mesh in the simulation. Here, we present an entirely new approach which resolves the above mentioned issues.

2. New multi-timescale methodology

Our innovation is to combine some of the machinery of time-stepped simulations, i.e., the finite differencing of equations and their solution on a spatial grid, with the time advance method of event-driven simulations. Discrete event simulation has the advantage of modeling system time advance through the use of irregularly time-stamped "events" that only update *what* needs to be updated *when* that needs to be updated. Historically, the fields of continuous and discrete event simulation have been largely distinct, with limited cross-disciplinary interaction (simulations mixing continuous and discrete models, e.g., to model circuits, is a notable exception). Time-stepped simulations have traditionally been used in continuous simulation to model physical systems described by partial differential equations and particles. On the other hand, event-stepped simulations have their origins in operations research and management science, and more recently have found application in war games and telecommunications [4]. Our goal is to combine state-of-the-art strategies and techniques of these two fields of simulations in order to achieve a breakthrough in technology that enables use of both spatial and temporal meshes and eliminate unnecessary computations. Here, we report on the first set of results obtained from this endeavor. In order to demonstrate the feasibility and advantages of this new approach, we chose a PIC model from plasma physics. Plasmas have a large number of states and highly optimized time-stepped codes have been developed to simulate them. As such it provides an excellent test bed for our new methodology.

3. Problem description

We consider the well-known problem of charging a spacecraft immersed in a neutral plasma by emitting a charged beam from its surface [5]. This problem is sufficiently complex to make a good subject for a feasibility study. We used a 1D (gradients only in one direction) simulation model in spherical coordinates in order to obtain the correct radial dependence of the electric field. We assume spherical symmetry so the only variations are with respect to radial distance from the spacecraft. Fig. 1 illustrates the simulation geometry. The spacecraft is taken to be a spherical conductor immersed in the charge-neutral plasma of the solar wind. The solar wind is taken to be stationary here. A charged beam is injected from the surface of the spacecraft at regular time intervals *InjectPeriod* with a density, *BeamFrac* (normalized to the electron density in the solar wind). The spacecraft is initially assumed to be charge neutral. However, each time the beam is injected, the spacecraft accumulates a charge equal in magnitude but with an opposite sign to that of the injected beam. As the beam injection proceeds, the spacecraft charges up negatively in case of a positron beam or positively in case of an electron beam. If the spacecraft is charged negatively, it repels

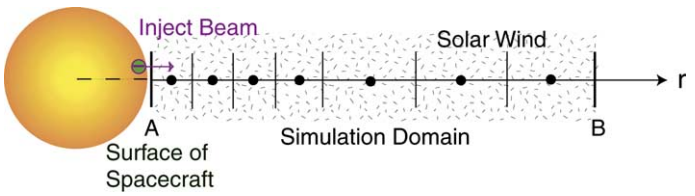

Fig. 1. Schematic of the 1D simulation model used in benchmarking our new code.

the solar wind electrons from the vicinity of the spacecraft and starts to pull in the beam and solar wind ions. Since solar wind ions are more massive than positrons the beam particles respond to the charged spacecraft earlier than the background ions.

All simulations are one-dimensional in configuration space [1]. In order to correctly account for the drop off of the electric field in 1D, we utilize the spherically symmetric Poisson equation:

$$1/r^2 d/dr(r^2 E_r) = 4\pi\rho \tag{1}$$

The solar wind plasma has a uniform density. We load the solar wind plasma with a uniform distribution. Therefore, charge particles in shell j are taken to have a charge given by $Q_j = 4\pi r_j^2 \Delta r_j \rho_j$ in order to be consistent with a uniform density. We will use normalized units throughout, where time is normalized to electron plasma frequency ω_{pe}, r is normalized to electron debye length, and velocity is normalized to electron thermal velocity. The simulation parameters are $T_{e\text{-sw}}/T_{i\text{-sw}} = 2.4$, $T_{e\text{-beam}} = 0$, beam energy of 10 kev, $m_i/m_e = 1836$ and a beam density so that at injection period of $\omega_{pe}\tau_{inj} = 0.004$, $N_{beam}/N_{sw} = 0.025$. Here, $T_{e\text{-sw}}$ and $T_{i\text{-sw}}$ are the solar wind electron and proton temperatures, $T_{e\text{-beam}}$ is the beam temperature, and N_{beam} and N_{sw} are the density of the beam and solar wind, respectively. We have considered both electron and positron beam injections. In the case of the electron beam, the solar wind ions are mostly unaffected whereas with a positron injection, the ions close to the spacecraft get pulled in towards the spacecraft. In the latter case all three species contribute to the evolution of the system, making it more complex. Therefore, we show results only for the positron beam injection although we have also done detailed benchmarking with electron beams with similar success. The parameters used here are not meant to correspond to actual experiments performed in space and are simply used as a way to test the viability of our new methodology.

4. Computational model

At any given time the global state of the electrostatic plasma system described above is fully characterized by the states (positions and velocities) of all simulation particles and a set of predefined beam injection rules. There are at least two ways of solving for the electric field. In TDS, the electric field is found by collecting the grid charge from particles and solving the Poisson equation with appropriate boundary conditions. In DES, the field is solved only locally and the electric field is found by keeping track of the charge that crosses a boundary, and updated using the differential form of Gauss's law $dE \propto dQ/r^2$. The progress of time can be modeled either by advancing particles with a *global* time step (''time-driven'' simulation) or by executing particle ''events'' (instantaneous state changes) at irregular time intervals determined through *local* physics considerations (''event-driven'' simulation). The former represents the conventional approach to modeling natural systems, the latter serves as an example of our new multi-time scale methodology. Both types of simulation are grid-based and use the concept of a global time clock that controls the progress of simulation time. For simplicity, we deposit particle charges on the grid via the nearest grid point (NGP) interpolation scheme [1]. The NGP scheme is not commonly used in particle simulations because it is noisier than first or higher order weighting schemes [1]. In 3D simulations, however, the extra computations needed for the higher order weighting schemes can become significant and in such cases NGP becomes a good alternative. In the present case, we have run our TDS code both with NGP and linear weighting and have found no significant difference in the results. So we only consider the NGP scheme in our DES code. Higher order schemes can also be implemented in DES but would leave to additional overhead as they do in TDS.

Below we describe the most essential features of both methodologies.

4.1. Time-driven simulation (TDS)

The algorithmic flow of a typical time-stepped electrostatic simulation cycle is shown in Fig. 2(a). The time-driven execution model is characterized by an iterative loop where all the future particle states (coordinates and velocities) are calculated from the present states *simultaneously* following the evaluation of static electric field in *all* shells (in our model this is done via the Gauss's law). All particles are pushed in time with the time step small enough to capture correctly both the beam and background plasma dynamics. Therefore, the time-driven simulation follows many clock "ticks" in which no significant physical changes in some parts of the system occur. From the programming viewpoint (here and below we use the terminology used by C/C++ and Fortran 90), the electric field is represented by an array of floating point numbers in TDS and shell objects in DES, and particle information is stored in an array of *Particle* structures. Each *Particle* structure (from now on simply a particle) holds in memory all the attributes of a single simulation

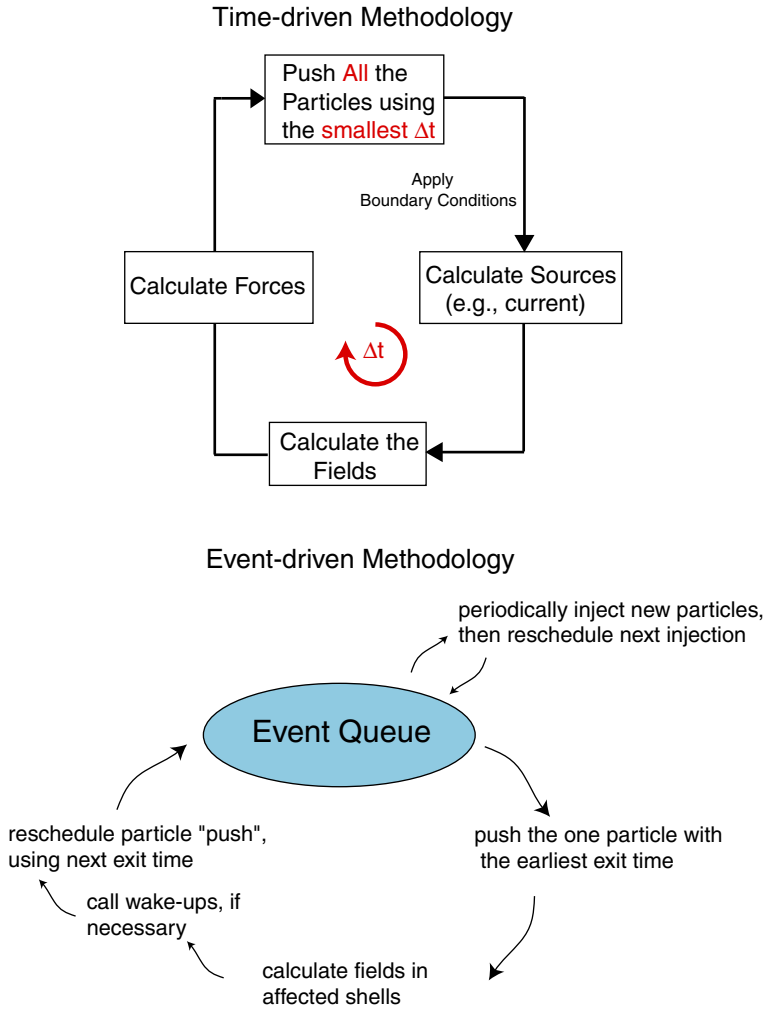

Fig. 2. Flow diagram for the time-driven and discrete event simulations.

macroparticle. Updating the system information at each time level is done by separately looping through the field and particle arrays. The system update is usually accompanied by collecting appropriate diagnostics. A well known constraint of TDS is the CFL condition. Any advance of the system using a time step larger than the CFL condition leads to numerical instability [1]. As we will demonstrate shortly, this is a very stringent condition and is at the root of the inefficiencies in time-driven simulations.

The field equation in Fig. 2(a) is solved by finite differencing equation (1):

$$4\pi r_j^2 \Delta r_j \rho_j = r_{j+1/2}^2 E_{j+1/2} - r_{j-1/2}^2 E_{j-1/2}, \tag{2}$$

where $r_{j-1/2} = \frac{1}{2}(r_j + r_{j-1})$ and $r_{j+1/2} = \frac{1}{2}(r_j + r_{j+1})$.

4.2. Discrete event-driven simulation (DES)

In an event-driven simulation, the temporal dynamic behavior of the system is modeled by the occurrence of "events". An event is modeled as a structure that contains a time stamp (event execution time) and a process function associated with a change scheduled to take place in the system being simulated. In mesh-based physical systems most events are shell based, i.e., a typical event reproduces a certain aspect of the complex physical behavior of the system on a *shell scale* (in order to underline the spherical symmetry of the problem we will be using the term "shell" instead of the more general "cell" concept).

The present time is the time stamp of the current event being processed. The simulation time jumps from the present time to the time stamp of the next event. Each event can schedule new events for future execution or retract pending events (ones that have been scheduled but not yet executed) based on the currently available information. Therefore, parts of the system can evolve on their own time scales. Obviously, this complicates the programming logic of an event-driven simulation, making it "fuzzy" as opposed to the "transparent" logic of a time-driven simulation. The discrete event algorithm implemented in our electrostatic model is as follows (Fig. 2(b)).

1. Initialize the simulation with given fields and particle distribution.
2. Using the known field values, compute particle future move times (the move time is the next time at which the particle will be moved, either across the shell boundary, or some projected distance in space). The simulation uses a structure of nested particle queues (see below).
3. The global event queue will choose which shell to process next, and the local queue will choose which particle within that shell to process.
4. Track the "source" term (e.g., charge enclosed by a boundary) for the fields as the particles pass from shell to shell. In this (electrostatic) case, it suffices to track the particle density but in electromagnetic simulations, there exist waves that are noncompressional and thus other (field) variables are needed in addition to density.
5. If the field changes above a certain threshold in a given shell, it is woken up. Advance the local particles in that shell up to the present time, calculate their new move times and update the affected shell queues accordingly. Note that the field updates are limited to the active shells (ones that have been allowed to schedule events) and their immediate neighbors. There are two options when dealing with a particle that enters a new shell when it does not trigger a wakeup. First, we can calculate the field in that shell and use that value to compute the exit time for that particle. Second, we can use the existing value of the field in the shell and use that for the computation of the exit time. In the present case, the local field computation is trivial and does not take much computational power. However, for more complicated field equations, the first option would be more CPU intensive. We have verified that both options yield identical results as long as the wakeup threshold is set appropriately. The second option is computationally more efficient and is being used in electromagnetic codes that we are developing.

6. Repeat 3–5 until the end of the simulation is reached. In most cases, the end time stamp is specified at the beginning of the simulation. Termination condition is either end time (the next event to be processed has a time stamp greater than the end time), or when any beam particles reach the end of the allocated shells.

Causality is assured by requiring that (i) the time stamp of any new event be at least as large as the current time of the simulations and (ii) the simulation executive always processes the events in time stamp order [4]. Also note that unlike implicit schemes which by necessity introduce artificial damping [3], the equations of motion here are time-reversible as they are in standard explicit codes. Comparing the algorithm for DES and time-stepped simulations, several advantages of DES methodology become immediately evident:

- *Local field computation* – In time-stepped simulations, the field is updated in the entire simulation domain at each time step. In DES, however, field is only updated in shells where a wakeup has been triggered. This is a major advantage in large three-dimensional simulations [6] where the number of cells can be tens or even hundreds of millions.
- *Natural temporal decoupling of particle species and fields* – In time-stepped simulations all particle species (e.g., electrons and ions) as well as fields are generally advanced based on the same time step. Techniques to decouple field and particles such as subcycling have been introduced but such techniques are usually cumbersome and do not always produce reliable results. In DES, however, the temporal advance of each particle species and the fields are naturally decoupled. Furthermore, even within each species, each particle is advanced based on its own exit time (time step). One can think of the DES simulation as having a complex temporal mesh consisting of an arbitrarily large number of layers and with each mesh having the capability to adjust its width.
- *Breaking the curse of CFL condition* – In a time-stepped code, the time step is limited by several factors: (i) frequency of underlying modes such as plasma frequency, and (ii) two types of CFL conditions, one on the fields and one on the particle. The time step has to be small enough so that the fastest mode/particle in the system cannot cross a shell in one time step. In an inhomogeneous system, the CFL condition can vary significantly across the simulation domain but in the time-stepped methodology, the local CFL condition with the smallest time step sets the global time step. This is clearly very inefficient. In DES, as we will show shortly, the CFL restriction is removed.
- *Efficient diagnostics and data storage* – In DES, one can easily generate a global knowledge of the simulation by advancing all events to a specified time for diagnostic purposes. One of the challenges with large three-dimensional simulations is the handling of large data files that are generated. This problem becomes even more acute with particle simulations as the number of particles can exceed the number of cells by a factor of 10 or more. DES offers a natural way to save only the part of the data that has changed rather than the whole dataset as is done in TDS. In 3D, this can lead to significant improvements in I/O as well as reduction in data storage requirements.

The complete discrete event simulation program is built of several software components as we now describe.

5. Building blocks of our DES code

5.1. System state

The global system state is represented by a collection of DES shells, with particles sorted by the shell location. Each particle contains an individual time stamp that determines its next move time. Inside each

shell the particles (or pointers to particles) are stored, sorted by exit time, in a local container in the time stamp order using either the *priority queue* or *multiset* data structures (see the detailed explanation of these constructs in Appendix A).

5.2. Events

The temporal evolution of the simulation system is accomplished by processing a sequence of events. Pending events (events that have been scheduled but not executed yet) are pushed onto a global priority queue, implemented as either a heap-sorted dynamic array or a multiset. The simulation loop (engine) runs by continuously removing the smallest time-stamped event from the global queue and processing it by executing an event handling function that causes appropriate change to the system parameters and may schedule or retract (cancel) previously scheduled but still unprocessed events.

All event classes inherit from (i.e. share the functionality of) the *Event* base class, which defines a two member interface: (i) a virtual function, "Process()," and (ii) a floating point "ProcessTime". The engine treats all events identically regardless of the fact that different events in general contain different member data and different implementations of Process() (polymorphism in object-oriented terminology). "ProcessTime" sets the simulation time at which Process() will be called (i.e. the event will be processed). In our code, all events are implemented as C++ objects and characterized by the common *Process()* virtual function, which can be implemented differently for each event. Below we use the C++ class notation when discussing the functionality of this event handling procedure for different events defined in this simulation.

Shell::Process() executes in response an event associated with a shell update. A *Shell* object contains a local data structure (either a custom version of a priority queue, or a standard library (STL) multiset) which holds all the particles within that shell, sorted according to the particles' move time. In the present electrostatic code, the force experienced by a particle is constant within a shell and the motion of the particle can be predicted all the way across the shell, without sacrificing accuracy. Its "exit" time is calculated by finding the roots of the quadratic equations:

Right exit equation:

$$0 = \frac{1}{2} * Acc * dt^2 + Vel * dt + Pos - RightBoundaryCoordinate. \tag{3}$$

Left exit equation:

$$0 = \frac{1}{2} * Acc * dt^2 + Vel * dt + Pos - LeftBoundaryCoordinate, \tag{4}$$

which gives:

$$dt = \frac{-vel \pm \sqrt{vel^2 - 2 * acc * (pos - leftbc)}}{acc}, \quad \frac{-vel \pm \sqrt{vel^2 - 2 * acc * (pos - rightbc)}}{acc}. \tag{5}$$

Notice that it is possible that the equation roots become imaginary, which means the particle does not exit the shell in that direction. The smallest real value for dt determines the exit direction and exit time of the particle. Alternatively, each particle can be assigned an arbitrary dt small enough for the particle not to cross more than one shell at a time. In this sense, the DES code can be thought of as selecting the most appropriate time scale for each particle at each step, transparently allowing for multiple time resolutions within the same simulation. For each physical shell, a *Shell* event is always scheduled for the earliest move time of all particles in that shell. Shell::Process() then moves the particle next in line to the top of the local priority queue.

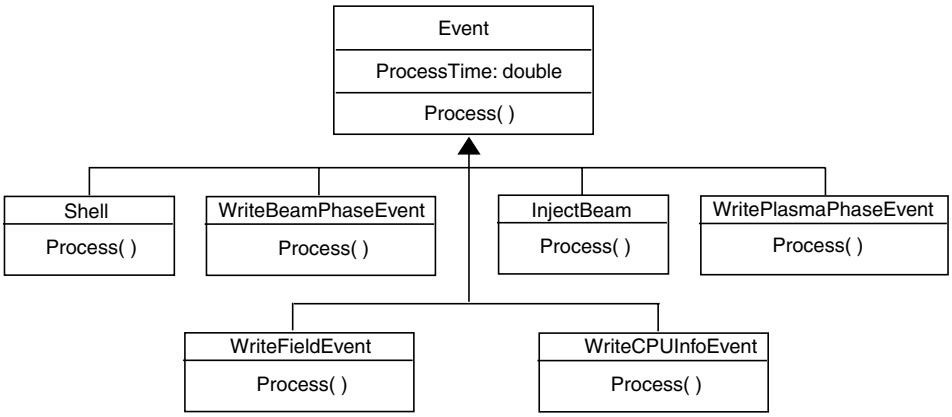

Fig. 3. UML diagram for the Event Class Hierarchy.

Because of the spherical symmetry in the simulation, the electric field at every shell boundary can be found by keeping track of the charge enclosed by that boundary, and applying Gauss's law, which in normalized units becomes:

$$E = \frac{Q_{enc}}{Factor * r^2}, \quad Factor \equiv 4 * PI * BackDens * Debye^3. \tag{6}$$

Thus, each shell keeps track of the field at its left and right boundaries by keeping a running sum of all the charges which have passed through that boundary. If the field within a shell changes by some user-defined tolerance, the shell will call a "wake-up" on all the particles which it contains. In a wake-up, the particles are pushed in time so that their positions and velocities are changed to current positions and velocities, using the old value of the field. Then, new move times are computed for all the particles using the new field value, the local particle queue is re-sorted, and a new event is scheduled based on the new earliest move time. This automatically retracts the old event.

A UML diagram for the Event Class Hierarchy is shown in Fig. 3. A brief description of each event follows.

InjectBeam::Process() injects particles periodically from the spacecraft into the plasma, updating the charge on the spacecraft and the field in the closest shell to reflect the change.

WriteBeamPhaseEvent::Process() periodically writes the phase of all of the beam particles to a file at a user specified interval.

WritePlasmaPhaseEvent::Process() periodically writes the phase of all of the plasma proton and electron particles to a file at a user specified interval.

WriteCpuInfoEvent::Process() is used to track the amount of CPU time used by the simulation, along with the current number of active shells and the number of events which have been processed.

WriteFieldEvent::Process() periodically writes the electric field vs. radius to a file.

5.3. Discrete event simulation environment (DESE)

The efficiency and causal correctness of a discrete-event simulation depend on how one schedules, processes and retracts events [4]. Here, we discuss our object-oriented programming implementation and provide a detailed account of the *Event* and *EventQueue* classes and their relationship.

All event classes inherit from (i.e. share the functionality of) the *Event* base class, which defines a two member interface: (i) a virtual function, "Process(()," and (ii) a "ProcessTime". The engine DESE treats

all events identically regardless of the fact that different events in general contain different member data and different code for Process() (polymorphism). "ProcessTime" sets the simulation time at which Process() will be called (i.e. the event will be processed).

Central to the simulation engine is the *EventQueue* class. Only one *EventQueue* object is initialized per simulation. The *EventQueue* object is responsible for keeping track of all the pending (scheduled) events, sorted in such a way that the *Event* object scheduled with the earliest ProcessTime can be easily found and accessed. To accomplish this, the *EventQueue* contains a Standard Template Library (STL) priority queue of *EventWrapper* objects.

EventWrapper is a simple class, containing only a pointer to the *Event* object it is wrapping, and *its own copy* of that event's ProcessTime. The *EventWrapper* class also defines the operators < (less than) and == (equal to), which allows the wrappers to be sorted according to the *EventWrapper* 's value of ProcessTime. The *EventWrapper* objects thus "wrap" pointers to events, allowing them to be sorted according their scheduled execution times.

The simulation begins when the "Run()" member function is called on the global *EventQueue* object. The event processing loop proceeds as follows:

1. Pop the next item off of the *EventQueue* 's priority queue. This gives a pointer to the *Event* with the earliest time stamp.
2. If the *EventWrapper* object's ProcessTime is different from the *Event's* ProcessTime, discard the wrapper and go back to the start of the loop (1). This simple rule enables a simple and efficient event retraction mechanism. If the ProcessTime stamp of an event changed between the moment that event was scheduled and the moment when it is being processed (i.e. the event has in fact been rescheduled during this time interval), the engine will ignore the "invalid" wrapper after popping it off the queue.
3. Update the global clock to the time of the pending *Event*.
4. Call Process() on the *Event* contained in the *EventWrapper*.
5. Repeat (1)–(4) as long as the *EventQueue* is not empty (i.e. contains unprocessed *EventWrapper* objects) and the end time of the simulation has not been reached.

The relationship between EventQueue and EventWrapper data structures is presented diagrammatically in Fig. 10.

6. Simulation results

6.1. Comparison of results

The time step in TDS is based on the CFL condition $\Delta t < \Delta t_{\mathrm{CFL}} \sim$ ShellWidth/V_{fastest}. The waves travel at ion acoustic speed whereas the beam is supersonic so V_{fastest} is the beam speed at injection which is $\sim$20.41. Using the shell width of 0.24, we then have $\Delta t_{\mathrm{CFL}} = 0.01175$. Since particles can get accelerated in time, one typically uses a smaller fraction of Δt_{CFL} in the simulation. Here, we use $\Delta t = 0.004$. Spacecraft has a radius of 500 and is located at the edge of the first shell, the simulation box spans the range $X = 500$–2180 and consists of 7000 shells. The solar wind plasma is loaded uniformly but with different weights, and there are 30 particles/shell/species initially and the simulation is run up to $\omega_{\mathrm{pe}}t = 60$. The beam is injected every $\omega_{\mathrm{pe}}\tau_{\mathrm{inj}} = 0.004$. However, it is possible to inject less frequently but a larger beam so that the total flux of beam particles will be the same. As we will show, the results are insensitive to the injection period as long as the total flux of beam particles injected is the same.

Fig. 4(a) shows the charge accumulated on the spacecraft as a function of time based on TDS (black) and DES (red). The agreement between the two simulations is excellent and is within noise levels. Initially the

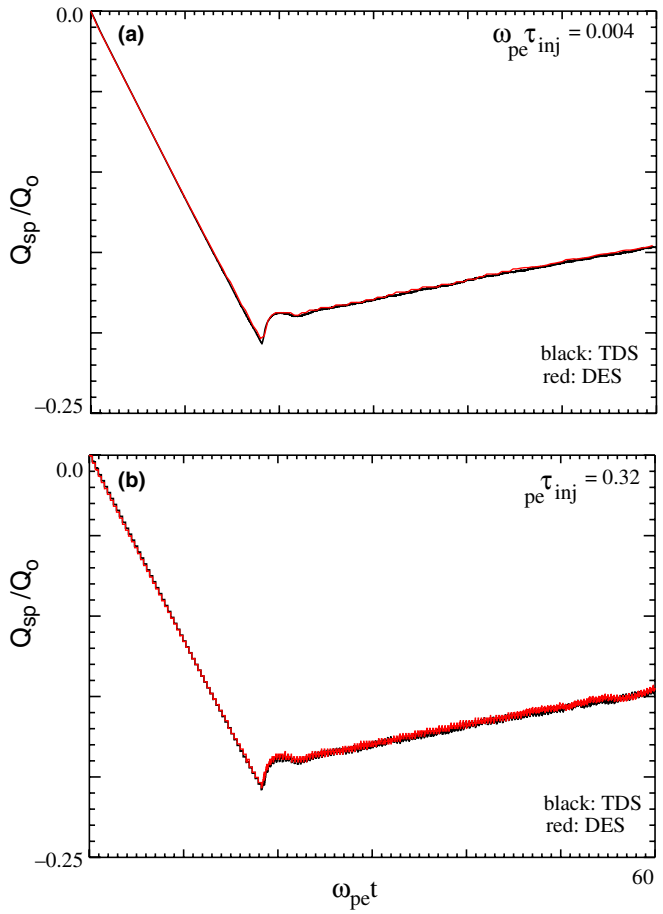

Fig. 4. Comparison of the time evolution of charge on the spacecraft from TDS and DES runs: (a) $\tau_{inj} = 0.004$ (run 1), and (b) $\tau_{inj} = 0.32$ (run 6).

spacecraft charge increases monotonically until the charge is large enough to pull in the electrons from the solar wind and repel the solar wind protons. At that point, the charge reaches a turning point and starts to decrease. Fig. 4(b) is similar to Fig. 4(a) except that now we have increased the beam injection period by a factor of 80 to $\omega_{pe}\tau_{inj} = 0.32$. The agreement between the TDS and DES remains very good and the results are similar to that in Fig. 4(a) except that now the spacecraft charge appears to increase in a step-like fashion. This step-like behavior is also present in Fig. 4(a) but it is on a much finer scale. The reason for this behavior is simple. Initially, and before the charge on the spacecraft is large enough to pull in particles from its surroundings, the spacecraft charge can only change when a beam particle is injected from its surface. Thus, the spacecraft charge remains the same between the injections, giving rise to a step-like evolution.

In order to compare the results from TDS and DES simulations in more detail, we next show in Figs. 5 and 6 the phase space from the two simulations for the solar wind electrons, beam particles, and the solar wind proton at the end of the run. The phase space data from the discrete event simulation was generated by bringing the global state of the system to the same time level as in the time-driven case. In order to show the dynamics close to the spacecraft, we have plotted in Fig. 5 the phase space as a function of log of radial distance. The behavior of the phase space further from the spacecraft is seen well in a linear plot of radial

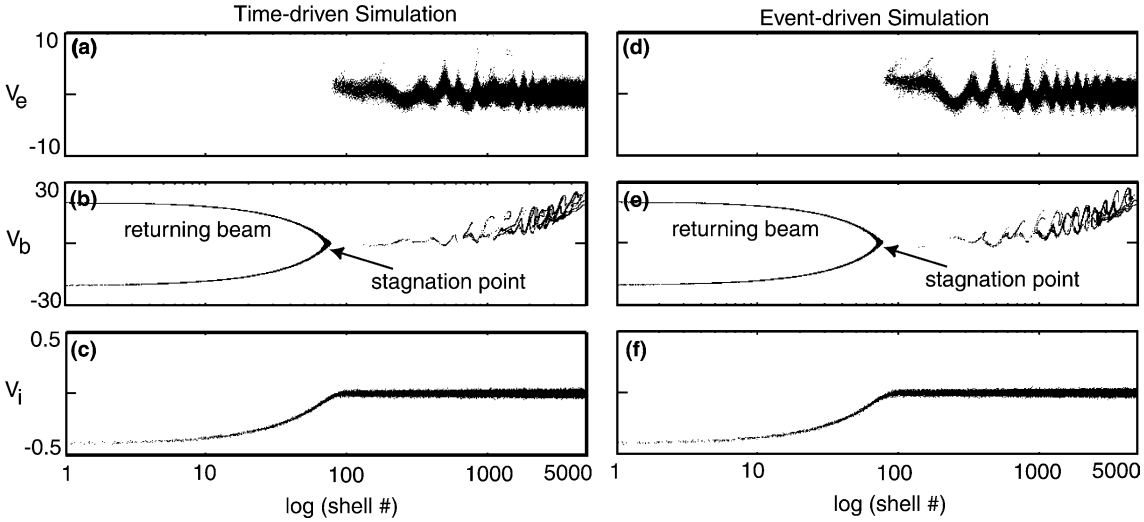

Fig. 5. Comparison of phase space structure at the end of the run for TDS and DES (run 1).

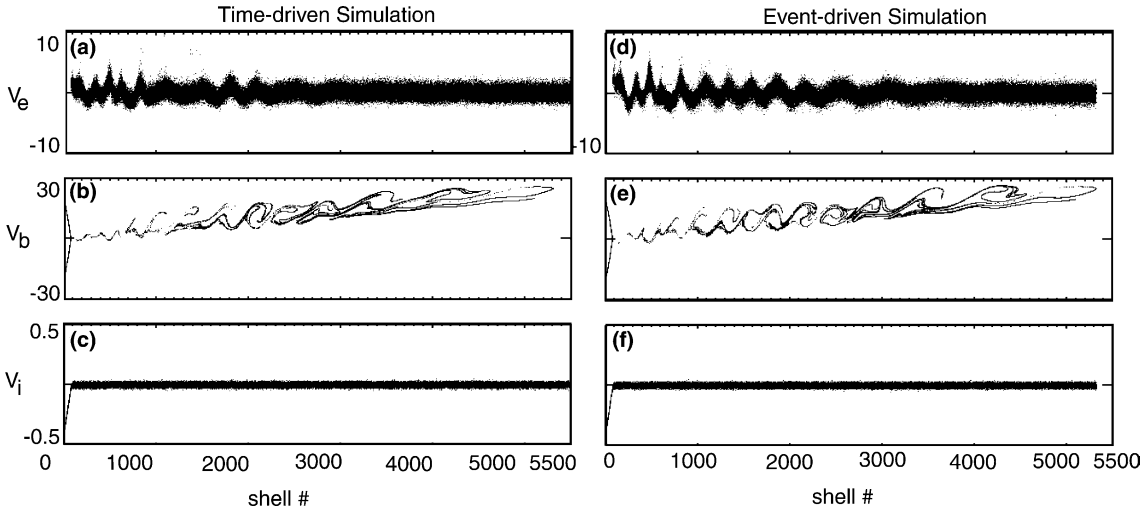

Fig. 6. Same as Fig. 5 except the x-axis (shell number) is in linear scale.

distance as in Fig. 6. Some of the beam particles reach the stagnation point (Fig. 5(b)) and then get pulled back to the spacecraft. Beam particles that can get past the stagnation point escape. The phase space of the electron beam (Fig. 5(a)) is empty in the vicinity of the spacecraft. This is because the spacecraft has developed a large negative charge at this time which in turn repels the solar wind electrons. Protons in the solar wind, on the other hand, are accelerated and pulled toward the spacecraft (Fig. 5(c)).

Comparing the phase space structure from the TDS and DES simulations in Figs. 5 and 6 reveal a number of points: (i) DES provides a very fine scale resolution in phase space. Looking at the beam fronts in this Figs. 6(b) and (e), we see that the beam has propagated to the same distance and has the same shape in both cases. (ii) Other main features of phase space (e.g., returning beam structure) for all species seem to

be in close agreement with the TDS simulation. (iii) However, the oscillations in electron phase space, due to the generated electrostatic waves, are more regular and better resolved (Figs. 5(d) and (e)) in DES as compared to the TDS (Figs. 5(a) and (b)). Similarly, the trapping vortices in the beam phase space are more coherent in DES (Fig. 6(e)) as compared to TDS (Fig. 6(b)). We have verified that this is due to two factors: (i) DES refines the time steps locally based on the underlying physical process and hence can have a higher temporal resolution than in the TDS. We will demonstrate this point in the next section. (ii) DES simulations are less noisy than the TDS. If we increase the number of particles in our TDS simulation and use a smaller time step, we would then get a phase space more closely matching that of DES shown in Fig. 5.

6.2. Breaking the CFL condition: Physics driven vs. numerically driven timestep

In the previous section we validated our methodology by comparing the results with the TDS. Although the two codes produced similar results, the computation that went into each code is quite different. In this section, we describe the differences in the temporal updates between DES and TDS. As we mentioned earlier, in TDS the time step is set by the global CFL condition. Strictly speaking, CFL condition is usually reserved for the field condition and requires that the time step be small enough so that no waves travel more than a shell width in one time step. In a particle simulation, there is also a condition that no particle should travel more than a shell width in one time step. Here, we use the term CFL condition interchangeable for both particles and fields. We emphasize that the CFL condition is not a limit based on the propagation speed of any real information in the system but rather it is based on the fastest speed that any information can theoretically propagate in such a system. For example, let us consider an initial configuration that is free of any disturbances and assume that the fastest allowable speed in the system is the speed of light. CFL condition requires that we take time steps small enough that the speed of light does not cross a shell in one time step. This restriction applies even if there are no light waves present in the system and violation of this condition leads to an exponentially growing numerical instability. The simple fact that light waves can exist in the system would require the system to be updated based on a speed of light CFL condition.

In DES, however, the system evolves only when a change has occurred and the time scale is highly individualized and is determined by a change in the system rather than the numerically driven time step in TDS. In TDS, the user must choose a time step a priori based on the numerical stability condition, $\Delta t < \Delta t_{\mathrm{CFL}} \sim \mathrm{ShellWidth}/V_{\mathrm{fastest}}$. In DES, the user does not assign a Δt but rather chooses a threshold condition which is used to signal a change in a particular location in the system. In this way the system sets its own time scale for change, which will vary both from shell to shell as well as during the run and can be much larger than the CFL condition, rather than a user imposed time step.

To illustrate this point, we note that in our present simulation, there are four physical entities of interest: the electric field, solar wind electrons, beam particles, and solar wind ions. These entities have their own temporal scales. In case of the electric field, the CFL condition is that the wave does not propagate more than one shell in one time step. Similarly, no particle can cross a shell in one time step. Thus each particle within a given species will have a different CFL condition. In TDS, all these timescales are merged together, and updated based on the smallest CFL. However, in DES, all timescales are naturally decoupled down to the single particle level, as we now demonstrate.

6.2.1. Field update

The field in a given shell is updated locally only when its value exceeds a pre-specified threshold. This causes a wakeup where the exit time of particles in that shell are recomputed based on the new field value. The time between each subsequent wakeup in a given shell can be thought of as the temporal resolution for the field. Figs. 7(a)–(c) show the minimum, maximum, and the average wakeup time as a function of shell number over the course of the simulation run of Fig. 4(a). The wakeup times are normalized to the CFL condition for the TDS where $\Delta t_{\mathrm{CFL}} = 0.01175$. Wakeup time is seen to have a complex dependence on the

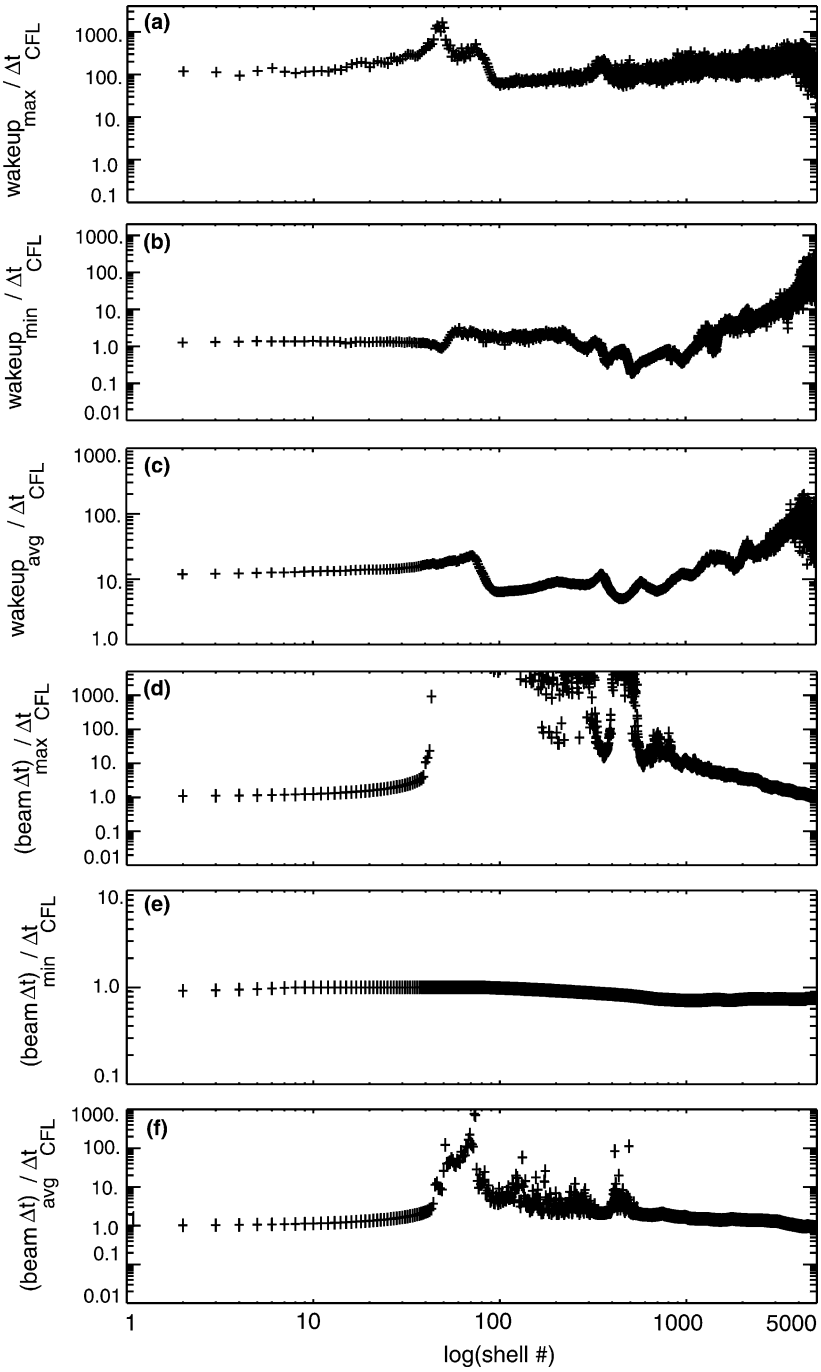

Fig. 7. Maximum, minimum and average of field wakeup and beam particle time step vs log of shell number for run 1.

distance from the spacecraft with the highest resolution varying between ~0.1 and 1000. DES allows the shells to update the fields only when necessary, instead of always updating at at a time that will be guaranteed fast enough (the CFL). Note that the fields in the DES simulation, in general, are updated less

frequently than in the time-driven code. This results in speed-up by removing unnecessary computation. Note also that sometimes the fields are updated more frequently than in the time-driven simulation, which results in greater accuracy in those shells. This intelligence is accomplished using only one parameter, a Wake-up tolerance (maximum allowed field error in that shell).

6.2.2. Particle update

Since in our algorithm a particle is moved, based on its exit time, one shell at a time, each particle satisfies its own individual CFL condition. This naturally decouples the temporal update of particles based on their velocity, mass, etc. For example, a proton is updated much less frequently than an electron. Similarly, a high energy beam particle is updated more frequently than a beam particle in the vicinity of the stagnation point. This is illustrated in Figs. 7(d)–(f), where we have plotted the minimum, maximum and the average time step for update of the beam particles versus the shell number over the course of the simulation. The minimum timestep is seen to dip below unity far from the spacecraft. This is because beam particles near the beam front (Fig. 7(c)) are the most energetic (largest V_b) and thus have a shorter exit time. Near the stagnation point, however, beam particles have a nearly zero velocity. As a result, the maximum timestep for such particles is really large (Fig. 7(d)).

In summary, the concept of timestep, which is central to TDS, is replaced by events. When an event is triggered in a given shell, that shell wakes up and an update is triggered. As we demonstrated, DES evolves each entity, down to an individual particle level, based on its own temporal scales and in the process eliminates the numerical stability issues related to the global CFL condition.

6.3. Performance comparison

In this section, we compare the performance of TDS versus DES. The timing of TDS is dominated by particles. In higher dimensions, the field solve can take up a significant portion of the total run time but in the present one-dimensional simulation it takes a small fraction of the CPU time. The execution time in case of DES, however, is mainly a function of the number of events that are triggered and processed. One of the advantages of DES is that is allows for relatively easy implementation of simulation features that would otherwise be difficult in TDS. For example, we have built in two features in our DES code that we can turn on and off as needed: (a) Expanding box which further avoids unnecessary computations by simulating only the relevant portion of the simulation domain. Rather than initializing the whole simulation box with solar wind plasma, it tracks the beam front and increases the simulation box in time as the beam propagates out and away from the spacecraft. (b) Two distinct spatial zones. It allows the user to break the simulation domain into two regions, each with its own grid spacing and threshold. The longer shells allow events to be further spaced and reduce the number of events.

Table 1 shows the performance of DES as a function of various parameters. All runs produce similar results. As is evident in Table 1, the two factors that significantly affect the performance of our DES simulation are the beam injection period and the use of larger shells farther out as they result in a reduction in the number of events processed. The expanding box in this case results only in a roughly a factor of 2 speed-up (run 9). Comparing runs 1 and 7 in Table 1, we see that the TDS performance is not affected much by reducing the injection period whereas DES CPU usage goes from 2425 to 114 s. This is because by scheduling a beam injection at every 0.004 timesteps, we are creating new events so frequently that it takes away from the performance advantage of DES. Since each particle uses as much CPU time as needed, and the beam particles move the fastest, they use by far the most CPU time. Thus, reducing the number of computationally intensive particles significantly improves the overall speed. As we mentioned earlier, this is unnecessary and a beam injection with $\tau_{inj} = 0.32$ produces same results (Fig. 4(b)) and it runs over 30 times faster than the TDS.

Table 1

Run #	τ_{inj}	FracBeam	#Near-shells	#Far-shells	Far-width	Threshold	CPU-DES (s)	CPU-TDS
Effect of threshold								
1	0.004	0.025	7000	0	0.24	0.08	2425	5395
2	0.004	0.025	7000	0	0.24	0.12	2317	
3	0.004	0.025	7000	0	0.24	0.3	2320	
Effect of τ_{inj}								
4	0.012	0.075	7000	0	0.24	0.12	793	
5	0.064	0.4	7000	0	0.24	0.12	242	
6	0.32	2	7000	0	0.24	0.12	140	
7	1.024	6.4	7000	0	0.24	0.12	114	4718
8	2.048	12.8	7000	0	0.24	0.12	112	
Effect of no expanding box								
9	0.32	2	7000	0	0.24	0.12	294	4726
Effect of two spatial zones								
10	1.024	6.4	100	1656	1	0.12	13	
11	0.32	2	100	1656	1	0.12	19	
12	0.064	0.4	100	1656	1	0.12	47	

Fig. 8(a) shows the cumulative CPU usage as a function of time for runs 1, 6 and 11 normalized to the equivalent CPU usage of TDS runs. The number of events processed as a function of time for these three runs is also shown in Fig. 8(b). In all cases, the speedup of DES over TDS is very large at early times where most of the activity is limited close to the spacecraft and the number of active shells is small. As the beam propagates further out, more shells become active, and the number of events increase. By the end of the simulation the beam has filled out almost the entire length of the simulation domain. However, in all three cases, DES is faster than TDS. In case of run 11, DES remains faster by more than two orders of magnitude over TDS. The number of events between runs 6 and 11 is initially identical but as the beam propagates to

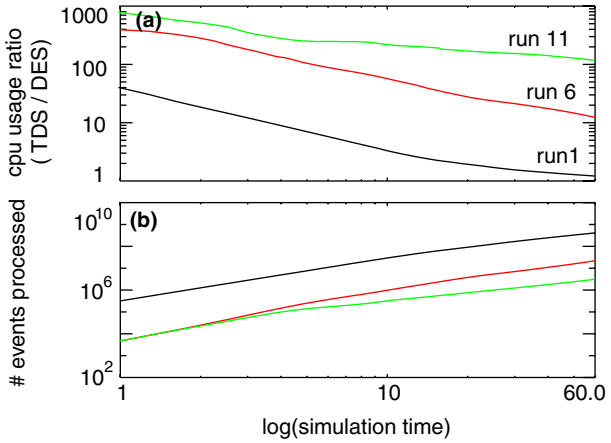

Fig. 8. (a) The ratio of cumulative CPU usage in TDS to DES for runs 1, 6, and 11. (b) The number of events processed as a function of time for runs 1, 6, and 11.

the outer shells, the number of events becomes larger in run 6. This is because we are employing larger shells for shells beyond 100 in run 11. In general, the performance advantage of DES over TDS is problem dependent. DES is suited for problems where the activity is localized and occurs on different temporal scales in different regions of the simulation. TDS is advantageous in problems where all regions are active and evolve on the same time scales.

7. Conclusions and future work

We have introduced a new simulation methodology which combines the spatial grid techniques of time driven simulations with the time advance scheme of discrete event simulations (DES). DES has the advantage that it decouples system element processing through the use of irregularly time-stamped events that only update *what* needs to be updated *when* it needs to be updated. Further, DES decouples parts of complex models by allowing them to evolve on their own simulated time scales. Discrete event simulation therefore supports a diverse model behavior in a more realistic and natural manner than time driven simulation. A key question was whether discrete event simulation can be used to model systems such as plasmas that have a large number of states. For simplicity, we limited our feasibility study to sequential execution of the so-called one-dimensional limit [1]. We were able to confirm that this was not only possible, but that DES plasma codes can be much faster (more than two orders of magnitude) than their existing time driven counterparts. However, the real power of DES is not just in performance but in the fact that it provides a natural infrastructure for developing intelligent algorithms that are ideally suited for modeling of multi-scale physical systems. For example, a major research topic within the TDS community is to get around the global CFL condition and allow for local time refinement. DES, on the other hand, naturally decouples spatial scales in time. Given the promise of this new technology, we are extending our work in several important areas: (i) We are developing a general parallel infrastructure for development of DES codes for modeling grid-based systems. (ii) In the electrostatic case, the field equation is elliptical and has no explicit time dependence. We have extended our technique to the parabolic field equation and have developed a code that solves the diffusion equation with inhomogeneous diffusion coefficient. We have also developed a parallel, electromagnetic hybrid (fluid electron, kinetic ions) code where the field equations have explicit time dependence. The details of these two codes (e.g., the flux conserving algorithm, treatment of ''steady state'' regions, etc.) will be presented elsewhere. (iii) We are developing a DES based multi-physics code. One of our main applications will be to global simulation of the Earth's magnetosphere. One final note is in regards to the accuracy and numerical stability of results obtained from discrete event simulations. We have calibrated our DES codes by comparing the results with analytical solutions in cases where a solution is known (e.g., in case of the diffusion equation) and comparison with results obtained from equivalent TDS. However, it is desirable to establish, for our grid-based DES technique, numerical stability conditions, effects of number of particles per cell, etc. as are done in traditional time-driven codes [1]. These questions are yet to be addressed and present an interesting research topic.

Acknowledgments

This research was supported by NASA SBIR Phase II contract at SciberNet, Inc. and by the National Science Foundation Information Technology Research (ITR) Grant No. 0325046 at SciberNet, Inc. Some of the computations were performed at the San Diego Supercomputer Center, which is supported by the National Science Foundation. Useful discussions with Professors Richard Fujimoto and Kalyan S. Perumalla are gratefully acknowledged. The authors also like to thank Jagrut Dave for useful comments on

an earlier draft of the paper and help with the UML diagram. Finally, we extend special thanks to the reviewer whose suggestions led to improvements in the presentation of the results.

Appendix A

The speed of the simulation is sensitive to the data structures used to implement the code. We looked at a number of different data structures to assess the impact cpu and memory usage.

Particles within each shell must be sorted according to their next MoveTime, the time at which the particle is to be pushed. This is usually the time at which the particle will exit the Shell, although the MoveTime may be set a number of different ways.

The C++ Standard Template Library (STL) contains a number of general purpose containers and algorithms. One of the simplest approaches is the use of the multiset container to create a sorted binary tree of particles. The nodes of the tree, which contain the particle data, are kept in a sorted order and linked via pointers. The particle with the earliest move time will always be stored in the left most node of the tree. An inorder traversal of the tree provides an ordering of the particles by increasing values of MoveTime. Fig. 9(a) shows the diagram of a collection of Particles is a multiset (binary tree), sorted according to their MoveTime.

While this approach does work, the multiset is not the fastest possible data structure for keeping items in sorted order. This problem becomes worse when a WakeUp is called on a Shell, at which point all of the particles will generally change their MoveTimes. When using a multiset, this means that all of the particles must be taken out of the tree, modified, and inserted one at a time into a new tree. If the MoveTimes were modified without first removing the particles, the tree would become unsorted and the data structure would

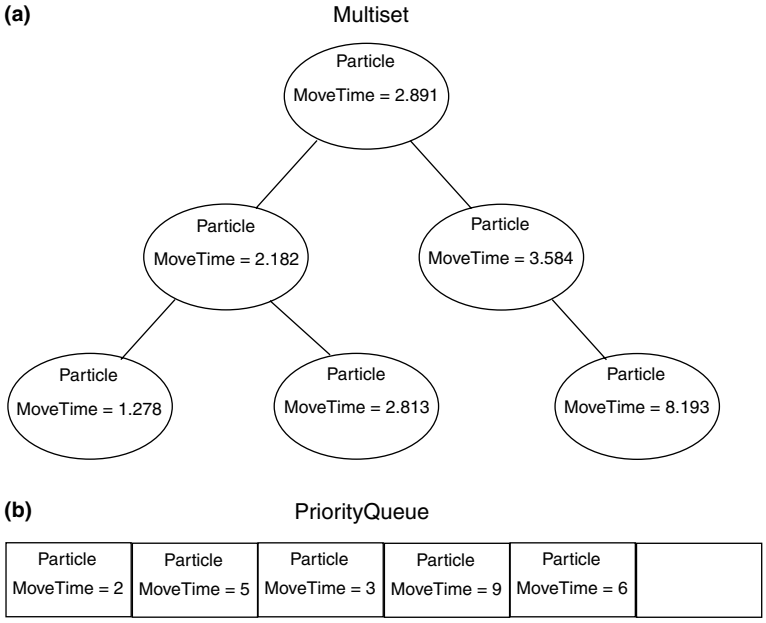

Fig. 9. (a) Particles in a multiset sorted according to their MoveTime. The multiset is the C++ standard template library version of a binary tree. (b) Schematic of the Particle PriorityQueue. The PrioriyQueue is implemented as a heap sorted array of Particles, which are sorted by their earliest exit time. Each Shell contains exactly one Particle PriorityQueue. This allows each Shell to schedule its next process for the earliest time a particle must be moved (usually from one shell to the next).

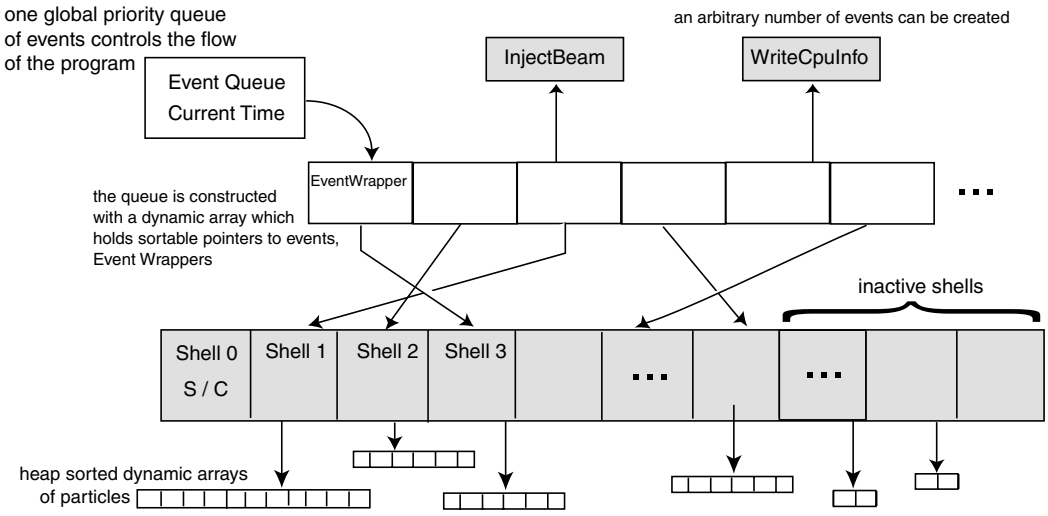

Fig. 10. Snapshot of simulation in memory.

be corrupted. It may be possible to write an algorithm to re-sort the tree, but this cannot be done with the STL multiset.

Instead, we created a custom data structure, the PriorityQueue, based closely on the STL priority_queue. The PriorityQueue is based on a heap-sorted dynamically resizable array, and contains all the typical priority queue functions. Heap sort performs with a worst case complexity of $O(N \log N)$. For large N, the complexity is equivalent to $O(N)$ as $\log N$ becomes nearly constant. Its performance in the average and worst cases are not very different and it requires fixed storage capacity. The Binary Tree approach described above requires a worst case sorting time of $O(N^2)$ and hence heap sort was preferred. Particles are pushed onto the queue by calling Push(), and the member with the highest priority (earliest MoveTime) can be looked at using Top() and removed using Pop().

The PriorityQueue differs from the STL priority_queuein two key respects:

1. The array in which the PriorityQueue stores its members is can grow and shrink dynamically. The stl priority_queue,for instance, can only grow. The ability to reduce the array size when many elements become unused is crucial for obtaining reasonable memory performance.
2. The members of the PriorityQueue can be accessed and modified directly using the bracket [] operators. This feature is used whenever WakeUp() is called on the shell. Modifying MoveTime of a particle while it is stored in the PriorityQueue will in general cause the PriorityQueue to become unsorted. For this reason, the PriorityQueue contains a Reset() function which resorts the members.

Fig. 9(b) shows a diagram of a PriorityQueue of particles, arranged in the same order in which they are sorted in memory. Notice that lower numbers have a higher priority. This is the desired behavior because we want particles with an earlier MoveTime to pop off the queue first. However, this is opposite from the behavior of a from a typical priority, which gives higher values a higher priority. The behavior was obtained by overloading the less than operator (operator<) to behave like the greater than operator (>operator) instead.

Fig. 10 shows a snapshot of the simulation in memory. Central to the simulation is the one global EventQueue, which stores pointers to EventWrapper objects in a priority queue, implemented with a dynamically

resizable array. An EventWrapper contains a pointer to an Event object and a data member EventTime which is the simulation time at which the event is processed.

Central to the simulation engine is the EventQueue class, which keeps a heap-sorted array of ''EventWrappers'', which are pointers to events sorted by earliest ProcessTime. The building block of the simulation are the Shell events, which are held in an array. Each shell holds a number of particles, which are again sorted by the exit time of the particle. Alternatively, the shells can hold sortable pointers to the particles, which reside elsehere in memory (such as a global particle array). Some events, such as InjectBeam and the diagnostic events (WriteCpuInfo, WriteBeamPhase, etc.) are only instantiated once in memory. The Shell Events are instantiated many times, and form the bulk of the simulation. Note, however, that not all of the Shells are active. Before a beam particle reaches a shell, it does not interact with other particles, and therefore does not scheduled events.

Each shell stores the particles it contains in a custom PriorityQueue class. The PriorityQueue is sorted so that the particle with the earliest move time has the highest priority. Again, the PriorityQueue is implemented as a dynamically resizable array. As an alternate implementation (not shown), the Shells keep track of pointers to the particles, rather than particles themselves. Using pointers to particles did not result in a significant performance benefit, so given the added code complexity required its use is not warranted in this implementation.

A.1. Scaling with various data structures

The data structures discussed above do not scale linearly with size. Updating and re-sorting them take up a non-trivial amount of CPU cycles and memory Similarly with the PriorityQueue – the larger the queue, the more operations it will take to sort it (which occurs whenever a particle is added or removed from the queue).

We studied scaling with particle density – the number of particles per cell – to show that the scaling, though not linear, is still acceptable for large projects. This is shown in Fig. 11.

We examined three possible codes:

1. Local PriorityQueue (discussed above) – the particles are kept in sorted order within a shell, using a custom PriorityQueue class. Events affect the entire shell.

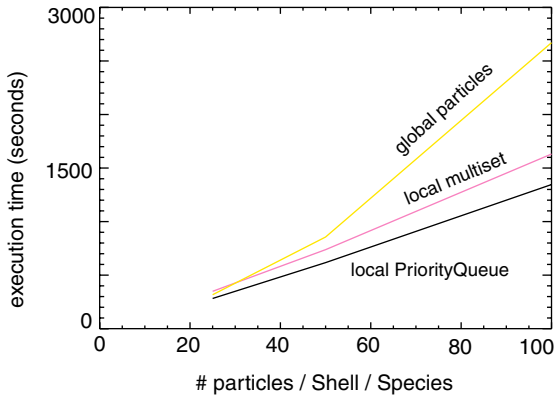

Fig. 11. Seconds for completion on a PIII 800 MHz, showing scaling of code speed with the number of particles (simulation time is 12 units).

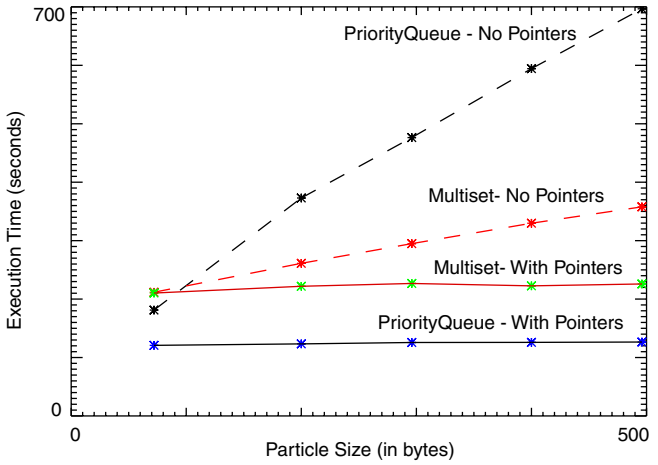

Fig. 12. Scaling of data structures with the number of particles.

2. Local multiset (discussed above) – particles are kept in sorted order within a shell, using a binary tree implemented as an STL multiset.
3. Events affecting individual particles– because the algorithm scales poorly with size, this discussion was not included in this paper. Briefly, the algorithm considers events as affecting individual particles, rather than each shell. For this reason, the number of events scheduled at any given time is equivalent to the number of particles in the simulation at any given time, rather than the number of active shells. This places excessive strain on the EventQueue, which must sort a much larger number of events. Further, when the field within a cell changes by enough to trigger a wake-up, *all* of the particles in that shell must be retracted and rescheduled, rather than just the one event associated with the shell itself.

A.2. Pointers to particles and particle size

Fig. 12 shows the scaling properties of a local PriorityQueue and multiset as a function of particle size. We ran the code up to 10 simulation times for this test. The multiset performance is unaffected whether pointers are used or not. The local PrirorityQueue, however, takes a performance hit for larger particle sizes when no pointers are used. But when pointers are used, it performs independently of particle size.

References

[1] C.K. Birdsall, A.B. Langdon, Plasma Physics via Computer Simulation, McGraw-Hill, New York, 1985.
[2] C. Dawson, R. Kirby, High resolution schemes for conservation laws with locally varying time steps, SIAM J. Sci. Comput. 22 (6) (2001) 2256.
[3] A. Friedman, S.E. Parker, S.L. Ray, C.K. Birdsall, Multi-scale particle-in-cell plasma simulation, J. Comput. Phys. 96 (1991) 54.
[4] R.M. Fujimoto, Parallel and Distributed Simulation Systems, Wiley–Interscience, New York, 2000.
[5] P.L. Pritchett, R.M. Winglee, The plasma environment during particle beam injection into space plasmas: 1. Electron-beams, J. Geophys. Res. 92 (A7) (1987) 7673–7688.
[6] H. Karimabadi, N. Omidi, Latest advances in hybrid codes and their application to global magnetospheric simulations, in: GEM (available online at http://www-ssc.igpp.ucla.edu/gem/tutorial/index.html), 2002.