A New Approach to Modeling Physical Systems: Discrete Event Simulations of Grid-based Models

H. Karimabadi[1], Y. Omelchenko[1], J. Driscoll[1], N. Omidi[1], R. Fujimoto[2], S. Pande[2], and K. S. Perumalla[2]
[1]SciberNet, Inc., [2]Georgia Institute of Technology

Abstract. The traditional technique to simulate physical systems modeled by partial differential equations is by means of a time-stepped methodology where the state of the system is updated at regular discrete time intervals. This method has inherent inefficiencies. In contrast, we propose a new asynchronous type of simulation based on a discrete-event-driven (as opposed to time-driven) approach, where the state of the simulation is updated on a "need-to-be-done-only" basis. Here we report on this new technique, show an example, and briefly discuss additional issues that we are addressing concerning algorithm development and their parallel execution.

1 Introduction

Computer simulations of many important complex physical systems have reached a barrier as existing techniques are ill-equipped to deal with the multi-physics, multi-scale nature of such systems. An example is the solar wind interaction with the Earth's magnetosphere. This interaction leads to a highly inhomogeneous system consisting of discontinuities and boundaries and involves *coupled* processes operating over spatial and temporal scales spanning several orders of magnitude. The brute force method of using full particle models for global simulations of the Earth's magnetosphere, with electron scale resolution everywhere in the simulation domain, is computationally infeasible. Such a computation would require over 10^7 years on the fastest parallel computers [Karimabadi and Omidi, 2002] that are available today. The ideal global code would have to take full advantage of the fact that there are regions within the magnetosphere with different modeling requirements: resolve electron physics only in the thin layers in the magnetosphere where reconnection is operational, resolve ion scales in the regions where the boundaries are formed and a lower resolution everywhere else. Such a code does not exist.

We have taken a new approach to the simulation of such complex systems. The conventional time-stepping grid-based PIC models provide the sequential execution of synchronous (time-driven) field and particle updates. In a synchronous simulation the distributed field cells and particles undergo simultaneous state transitions at regular discrete time intervals. In contrast to this well known technique, we propose a new, asynchronous type of PIC simulation based on a discrete-event-driven (as opposed to time-driven) approach, where particle and field time updates are carried out in space on a "need-to-be-done-only" basis. In these simulations particle and field information "events" are queued and continuously executed in time in a manner similar to that employed in the theory of cellular automata (CA). The rational of this approach is not to try to describe a complex plasma system by using instance differential equations, but to let the complexity emerge by modeling interaction of adjacent plasma cells following elementary rules that reflect the underlying laws of physics. Event-driven PIC simulations automatically guarantee that the progression of the system progress over time captures important state changes without processing "idle" information.

As shown in Figure 1, a DES system can be broken into two components: (1) the models and (2) the parallel simulation executive that manages events and the progression of simulation time

[Fujimoto, 2000]. Development of next generation plasma codes requires innovations in both components.

2. DES Algorithms: Issues and Solutions

2.1 Time integration. Field equations are discretized in space in the conservation form.

Each computational mesh cell is assigned discrete states associated with the temporal evolution of local field and particle quantities. Transitions from one temporal state to another are called "events". Time integration of each field component is delayed by a time interval depending on the magnitude of its predicted rate of change. Particles are scheduled for advance in each cell based on their current velocities, local field magnitude and cell size. Each computation cell keeps a registry of increments to its original state (the state used for the prediction) caused by the neighboring cells and reschedules events (time advances) to earlier times if the cell state is significantly altered during the predicted time delay. The DES code programming architecture is drastically different from conventional (time-driven) codes. In particular, each mesh cell has a means of polling its neighbors and fetching global simulation information using its local data handlers. It is also aware of its role in establishing communication with remote (distributed) parts of the system or applying proper boundary conditions. A nontrivial problem is to preserve fluxes across mesh cell interfaces.

In explicit time-driven codes adjacent cells are always advanced with fluxes taken at the same time level. DES cells schedule themselves asynchronously and therefore special care must be taken to ensure that field quantities in cells with common interfaces are always integrated in time with identical fluxes across the common boundaries.

We have developed a library of C++ classes (SciDES) designed to provide a set of discrete-event software tools for implementing finite difference and particle-in-cell methods for the solution of coupled partial differential equations and equations of particle motion. SciDES standardizes fundamental data structures and algorithms for programming distributed time-dependent scientific models on block-structured computational domains and formalizes the most essential aspects of the distributed physics-based DES models in the form of a pseudo-distributed architecture. This pursues several goals. First, the SciDES API separates the computational physics algorithms from the communication issues by abstracting them into well defined concepts (C++ classes) and providing all the necessary "go-between" implementation details. Second, it fosters more efficient cooperation of computational physicists with computer scientists working on the distributed discrete-event engine algorithms since it allows substitution of pseudo-distributed plug-in modules by their MPI counterparts in a plug-and-play fashion without breaking the physics core of the code. In addition, the ability to run virtual distributed simulations on a single CPU enables testing various physical mechanisms that provide important insight into predictive properties of physics-based parallel discrete-event simulations. An example of our SciDES architecture, the class MP DES which abstracts the virtual multi-processor DES environment, is shown in Figure 2. In this diagram solid arrows are indicative of inheritance (the “is a” relationship), dark dashed arrows represent ownership (the “has a” relationship) and light dashed arrows mark class instantiation from template classes.

As a way of testing this new methodology, we show in Figure 3 a one-dimensional simulation of a fast magnetosonic shock. For this test, we developed a DES equivalent of a time-stepped resistive hybrid code. Figure 3 shows the comparison of the results between the traditional time-stepped and our event-stepped simulation. We have plotted the y and z (transverse) components of the magnetic field, the total magnetic field, and the plasma density versus x after the shock has separated from the piston on the right hand side. The match between the two simulations is remarkable as DES captures the (i) correct shock speed, and (ii) details of the wavetrain associated with the shock. This match is impressive considering the fact that the differences seen in Figure 3 are within statistical fluctuations associated with changes in the noise level in hybrid codes.

2.2 Parallelization.

The parallelization of asynchronous (event-driven) continuous PIC models presents a number of challenges. As in conventional (time-driven) simulations, it is realized by decomposing the global computation domain into subdomains. In each subdomain, the individual cells and particles are aggregated into containers, which are mapped to distributed parallel processors in a way that achieves maximum load balancing efficiency. The parallel execution of conventional (time-driven)simulations is commonly achieved by copying field information from the inner lattice cells to the ghost cells of the neighboring subdomains and exchanging out-of-bounds particles between the processors at the end of each update cycle. In contrast, in parallel asynchronous PIC simulations both particle and field events are not synchronized by the global clock (i.e. they do not take place at the same time levels throughout the simulation domain), but occur at arbitrary time intervals, which may introduce synchronization problems if some processors are allowed to get ahead in time of other processors (the "optimistic" approach) [Jefferson 1985]. As a result, a processor may receive an event message from a neighbor with a simulation time stamp that is in its own past, thus causing a causality error. On the other hand, parts of a distributed discrete-event simulation can be forced to execute synchronously with remote tasks corresponding to the neighboring subdomains (the "conservative" approach). If so, the parallel speed-up critically depends on the underlying domain decomposition technique and additional predictive ("look-ahead") properties of the simulation in question. Regardless of the approach taken, it is important to note that DES computations offer substantial efficiencies compared to conventional explicit time-driven simulations due to the reduction in the amount of computation that mist be performed.

The following are some of the important issues that must be addressed in parallel discrete event simulations of continuous systems:

- **Synchronization**: This is by far the paramount issue to be carefully resolved for achieving the best parallel execution performance. Broadly there are two approachescommonly used - conservative and optimistic.
    - **Conservative**: This approach always ensures *safe* timestamp-ordered processing. However, runtime performance is critically dependent on *a priori* determination of an application property called *lookahead*, which is roughly dependent on the degree to which the computation can predict future computations without global information. In one conservative approach, events that are beyond the next

    lookahead window are blocked until the window advances sufficiently far to cover those events. Typically the lookahead property is very hard to extract in complex applications, as it tends to be implicitly defined in the source code interdependencies. The appeal of this approach however is that it is one of the easiest schemes to implement if the lookahead is somehow specified by the application.
  - **Optimistic**: This approach avoids blocked waiting by optimistically processing the events beyond the lookahead window. When some events are later detected to have been processed in incorrect order, the system invokes compensation code such as state restoration or reverse computation. Since blocking is not used, the lookahead value is not as important, and could even be specified to be zero without affecting the runtime performance. While this approach eliminates the problem of lookahead extraction, it has a different challenge – namely, support for compensating code.
  - **Combination**: Sometimes it might help to have some parts of the application execute optimistically ahead (e.g., parts for which lookahead is low are hard to extract), while other parts execute conservatively (e.g., parts for which lookahead is large, or for which compensation code is difficult to generate). In such cases, a combination of conservative and optimistic synchronization techniques can be appropriate.
- **Load Balancing**: As with any parallel/distributed application, the best performance is obtained when the load is evenly balanced across all resources. In parallel simulation in particular, load imbalance can have a very adverse effect. This is because typically the slowest processor can hold back the progress of simulation (virtual) time, which in turn slows down even those processors which are relatively lightly loaded.
  - **Automated/Adaptive**: Automated schemes are preferable for load-balancing at runtime. These schemes vary with the particular synchronization approach used.
  - **Support Primitives**: In order to permit automated/adaptive load balancing by the system, it is important to provide appropriate primitives to the application, so that application-level entities can be moved across processors easily by the system in a transparent manner as needed.
- **Modeling and Runtime Interface**: To be able to decouple the implementation details of the parallel simulation executive from the application/models, it is best to define the model-simulation interface in an implementation-independent fashion. This not only helps avoid reimplementation of models whenever the engine is changed, but also permits experimentation with multiple synchronization and load-balancing approaches for the same application. Additionally, it enables engine-level optimizations to remain transparent to the application, so that the application-developer is not burdened or sidetracked with such issues during model development.

With the preceding issues in mind, we are carefully developing appropriate interfaces and implementations of our parallel execution engine. A brief description of our approach follows:

- The synchronization issue is being resolved by providing a transparent interface that does not mandate one synchronization approach over another. The underlying implementation is also being developed such that different model entities can chose different synchronization (conservative or optimistic execution style), as is most appropriate for them.

- The load balancing issue is being addressed by the use of an “indirect messaging” interface layer that decouples application entities from their processor mapping.
- The modeling and runtime interface is also kept abstract and flexible, so that radically alternative implementations can be implemented underneath the interface.

References

Fujimoto, R.M., *Parallel and Distributed Simulation Systems*. 2000: Wiley Interscience.

Jefferson, D., *Virtual Time,* ACM Transactions on Programming Languages and Systems, 1085. 7(3):pp. 404-425.

Karimabadi, H. and 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.

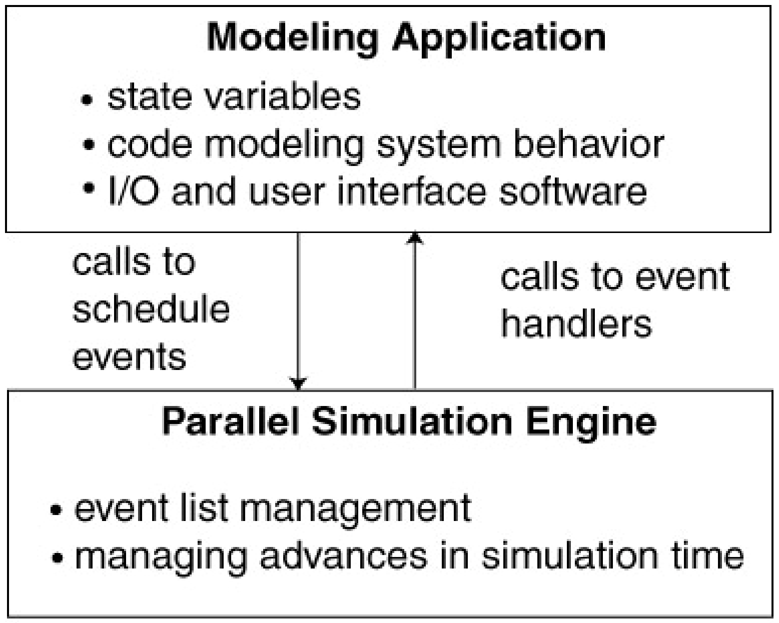
Modeling Application
• state variables
• code modeling system behavior
• I/O and user interface software
calls to schedule events
calls to event handlers
Parallel Simulation Engine
• event list management
• managing advances in simulation time

Fig 1. Components of discrete-event simulation

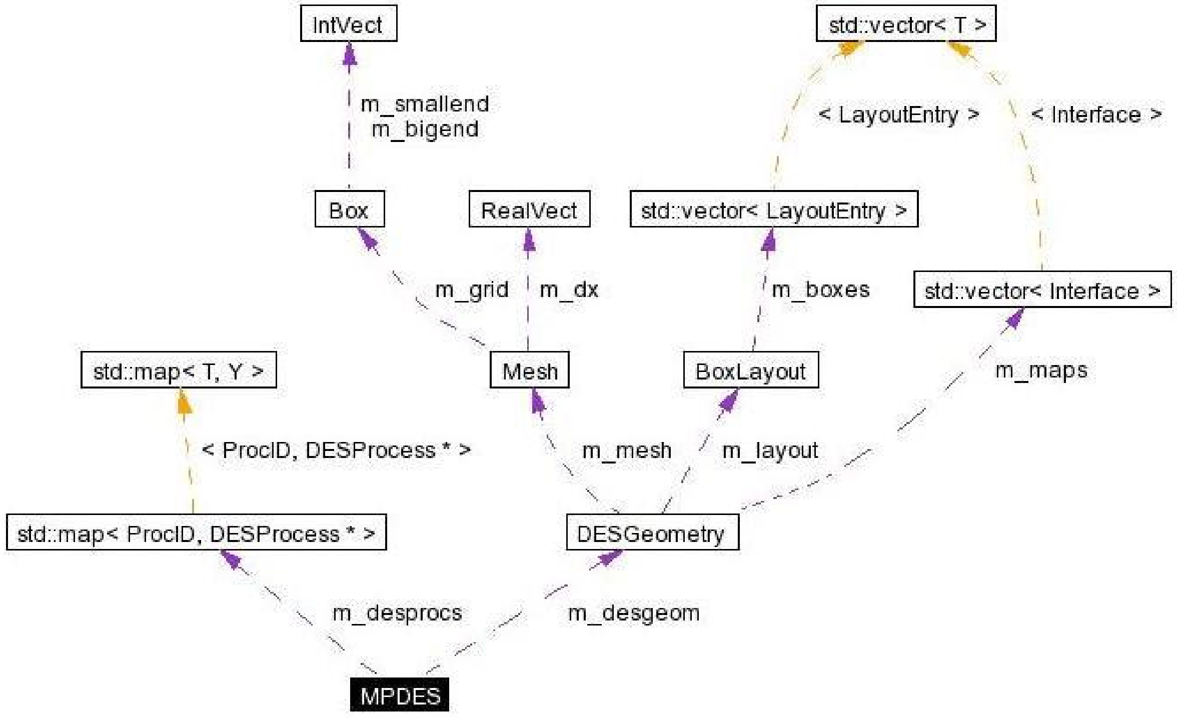

Figure 2. The *MPDES* class collaboration diagram. The MPDES object encapsulates the global simulation geometry properties and defines the table of virtual DES processes.

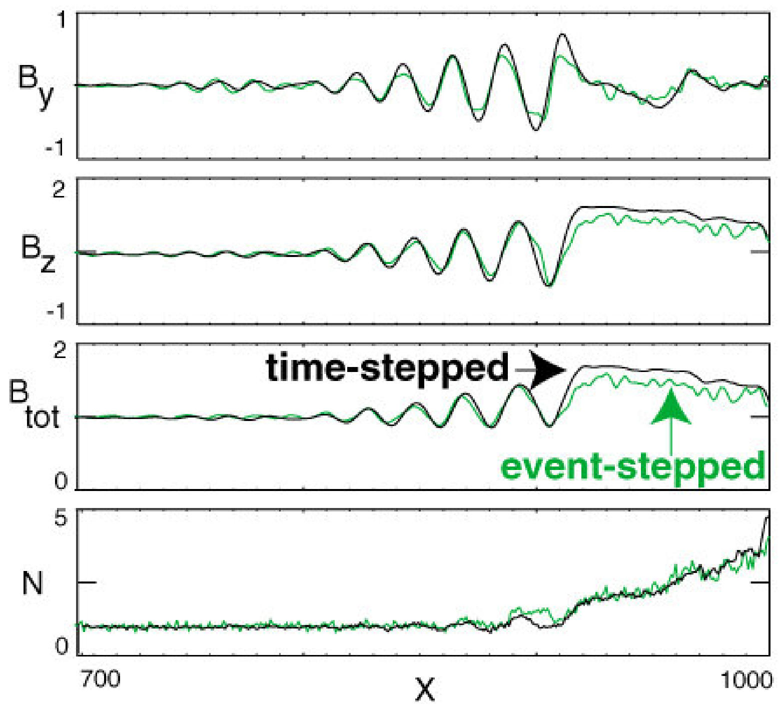

Fig. 3 - Time-stepped vs event-stepped simulation of a fast shock