# Optimistic Parallel Discrete Event Simulations of Physical Systems using Reverse Computation

Yarong Tang, Kalyan Perumalla, Richard Fujimoto *Georgia Tech*
Homa Karimabadi, Jonathan Driscoll, Yuri Omelchenko *SciberNet Inc.*

## Abstract

*Efficient computer simulation of complex physical phenomena has long been challenging due to their multi-physics and multi-scale nature. In contrast to traditional methods using time-stepped execution, we describe an approach using optimistic parallel discrete event simulation (PDES) and reverse computation techniques. We show that reverse computation can significantly reduce the execution time of a plasma simulation computation without requiring a significant amount of additional memory compared to conservative execution techniques. We describe an application-level reverse computation technique that is efficient and suitable for many complex scientific simulations involving floating point operations.*

## 1. Introduction

Parallel Discrete Event Simulation (PDES) has been an active research area in the high performance computing community for many years. Synchronization techniques for PDES systems are usually classified into two principal categories: conservative approaches that avoid violating the local causality constraint and optimistic approaches that allow violations to occur, but provide a mechanism to recover. The operation to recover a previous state in an optimistic parallel simulation is known as a *rollback*, and involves undoing incorrect computations.

A widely used technique for implementing rollback is *state-saving* that saves the values of state variables prior to an event computation and restores them by referring to these saved values upon rollback. *Copy state saving* creates an entire copy of a process's state, while *incremental state saving* keeps a log of changes to individual state variable. A relatively new technique for rollbacks, *reverse computation* [1], realizes rollbacks by performing the inverses of the individual operations executed in the event computation. These techniques have been exploited in small- or large-scale parallel simulations.

However, advances in PDES research to date have had little impact in space physical science, where multi-physics and multi-scale physical systems are modeled by partial differential equations and particles. Traditionally, simplified models of such physical systems have been simulated using time-driven or time-stepped approaches [2]. The inherent limitations of the time stepped approach prevents the simulation of more complex physical systems that are important in plasma physics and other areas of science. Even the latest techniques developed in time-stepped research such as adaptive mesh refinement (AMR) [3] are constrained by excessive computational requirements and necessity to update all cells within a given patch based on the Courant-Friedrichs-Levy (CFL) condition in that patch. Discrete event simulation was recently used to model spatially discretized physical systems [4] where it was shown that this approach not only alleviates the constraint of the CFL condition but also provides a significant performance advantage over the time-stepped approach.

In our work, we extend this work and apply optimistic parallel discrete event simulation techniques to this problem. Because memory constraints are often a severe limitation in the size of the computations that can be performed, reverse computation offers greater promise than traditional state saving techniques. We explore the use of reverse execution for plasma simulations to gain new insights for such challenging, complex physical systems. The combination of DES methodology and reverse computing techniques offer the potential to dramatically reduce the amount of time required to perform plasma simulations without incurring a large penalty in additional memory requirements.

The main contributions in this work can be summarized as follows. To our knowledge, this is the first work to apply reverse computation techniques to the parallel physical system simulations and to show performance advantages using this approach. In addition, we provide a simple model and guidelines for creating reverse simulation codes at the application level that can help physicists or astrophysicists to develop simulation prototypes without comprehensive knowledge of PDES mechanisms.

The remainder of this paper is organized as follows. The next section discusses related research. Section 3 provides an overview of the physical system we simulate and the reverse computation approach. Section 4 gives an in-depth discussion of the reverse computation implementation and discusses the challenges. Section 5 presents experimental results from a preliminary performance evaluation study. We conclude by reporting current and future work in this area and provide guidelines for reversing parallel physical simulation codes.

## 2. Related Work

A limited amount of research has examined physical

system simulation using parallel discrete event simulation techniques. Perhaps the earliest was the "colliding pucks" application developed for the Time Warp Operating System (TWOS) [5]. This work, modeling a set of pucks traveling over a frictionless plane, was used to benchmark an early implementation of the Time Warp protocol. Lubachevsky discusses the use of conservative simulation protocols to create cellular automata models of Ising spin [6]. Other work describes challenges in using discrete event simulation techniques for other physical system problems [7].

Seminal work in optimistic parallel discrete event simulation was completed by Jefferson [8]. State saving has historically been the dominant approach to enabling the rollback of computations. Use of reverse execution to roll back computations was first described in [1]. Their reverse execution procedures were automatically generated by a compiler. More recent work using reverse execution for parallel network simulations, using manually generated code, was reported in [9]. Our work is different in that it applies reverse execution techniques in simulating physical systems which involves complex floating point operations and generates reverse code based on application semantics.

Traditionally, complex physical systems described by partial differential equations and particles are modeled by time-stepped simulations which appear inadequate for today's complex physical systems in space physics. The authors in [4] recently demonstrated the feasibility of discrete event simulations to such complex systems and made a great effort in promoting the inter-disciplinary research in modeling and simulation. Our study is based on their work and further explores the feasibility of an advanced parallel synchronization mechanism in such systems – reverse computation

## 3. Overview

In this section we briefly describe the computational plasma simulation model used here. Details of the DES model of this system are presented in [4], and interested readers are referred to that work to gain familiarity with the physics and DES modeling methodology that were applied. Here, our focus is on use of reverse execution techniques in optimistic parallel simulation of this model.

### 3.1. Computational Model: PIC Simulation

One of the great challenges in space physics is to understand how the solar wind interacts with the earth's magnetosphere. The work presented in [4] was the first to use a DES approach to simulate such complex physical systems; there, a feasibility study based on the well-known particle-in-cell (PIC) model [10] was described to provide the foundation for development of such multi-physics and multi-scale simulations. This model is conceptually simple, but sufficiently complex for our feasibility study. Here, we limit our simulations to a one-dimensional electrostatic model using spherical coordinates.

Figure 1 illustrates a plasma PIC simulation of charging a spacecraft immersed in neutral plasma by injecting a charged beam from its surface [10]. The spacecraft is initially charge-neutral and immersed in the charge-neutral plasma of the solar wind. A charged beam is periodically injected from the spacecraft surface which then causes the surface charge to change.

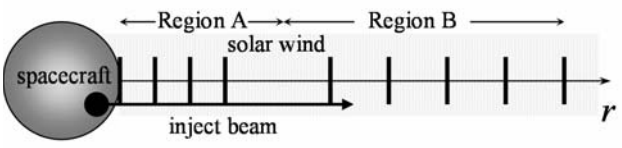


**Figure 1:** Schematic of the PIC model

### 3.2. Parallelization

In PDES of a PIC model, the simulation domain is divided into "cells" with each cell mapped onto one logical process (LP). In our spacecraft model, each cell is called a "shell", and is modeled as a Shell class (we will use the terms cell and shell interchangeably). Two distinct regions are shown in figure 1, based on different grid spacing. The state of each LP includes the cell field variables and the states of all particles within the cell boundaries. The dynamic behavior of the system is driven by particle movements that are modeled by events. Whenever a particle moves across a cell boundary, the electric field in the affected cells is updated by keeping track of the charge that crosses the boundary [4]. Note that the field updates are limited to active cells where events are allowed to be scheduled.

There are three types of events associated with particle movements: *ParticleArrivalEvent*, *ParticleDepartureEvent* and *ParticleInjectEvent*. The corresponding event handlers are outlined in figure 2. When the simulation starts, all active cells are initialized, including electric fields and particle states (velocities, positions, etc.). In particular, each particle's movement is determined by its *MoveTime* or cell exit time (a time in the future when the particle will exit the hosting cell). Whenever a particle is created or inserted in a cell, its exit time must be calculated and a pair of departure and arrival events scheduled at the exit time. In addition, a particle's exit time needs to be recalculated whenever the hosting cell "wakes up". A wakeup happens when a cell's field changes beyond a threshold value, resulting in recalculation of exit times of all its particles.

```
Shell::arrival( ParticleArrivalEvent *e ) {
      if  ( this cell is active ) {
        update cell state;
        insert particle in cell;
      } else if ( e is a beam particle ) {
        activate cell;
      }
}
Shell::departure( ParticleDepartureEvent *e ) {
      if  ( particle bounced from right neighbor ) {
        bounce particle back; // no cell state change
      } else {
        update cell state;
      }
}
Shell::inject( ParticleInjectEvent *e ) {
      update cell state;
      insert beam particles;
}
```

**Figure 2:** A simplified PIC model

### 3.3. Reverse Computation Approach

The characteristics of this plasma simulation present three major challenges concerning the synchronization of parallel computation. The rationale for the parallelization approach used here is based on the following considerations.

- **Lookahead.** The simulation is highly dynamic. The amount of parallelism can vary dramatically as the simulation progresses. Dependencies among events are governed by each particle's exit time, but this time can be arbitrarily close into the future. This low level of "predictability" results in a low, dynamically changing value of lookahead that makes efficient execution using conservative synchronization techniques difficult. This suggests that optimistic synchronization [8] may be a more natural choice for this simulation.

- **Memory**. Realistic plasma simulations involve large numbers of events, on the order of billions. Complex data structures are often needed. This makes traditional approaches to optimistic execution using state saving problematic: the amount of memory required can be prohibitively large. Further, the amount of computation performed for each event tends to be relatively small, on the order of microseconds on a contemporary CPU. This suggests that the time required for state saving may be significant, and hence could significantly degrade performance. Both of these factors suggest that reverse computation technique is a more suitable technique to realize an efficient optimistic execution of this plasma simulation code.
- **Floating point.** The reverse computation approach proposed in [1] uses an automated approach to creating the reverse execution code for each line of forward execution code. For example, a decrement statement is generated to undo an increment statement in the forward execution code. This approach becomes problematic when floating point arithmetic is used because the computation may not be easily reversed due to effects such as round-off error. Here, we explore a different approach where the program is viewed at a higher level of abstraction, and suitable reverse computation code is developed manually.

These factors motivate the approach that was adopted for optimistic synchronization using manually derived reverse computation code. We believe this can be used to build a foundation for our future work in developing scalable parallel simulators for complex physical systems.

## 4. Parallel Simulation Code

Here we use a one-dimensional model of the spacecraft electrostatic particle code as an illustrative example to discuss some of the challenges in generating the reverse code for this physical system simulation. There are two types of distinct physical entities in this simulation: particles and cells[1]. Particles move across cells and each cell keeps track of the particles residing within its own domain. The communication between adjacent cells occurs via particle movement events that contain information of particle physical states. The complex data structures housing the particles and physical processes being captured require a careful modeling of the system. The object-oriented design used here allows one to encapsulate physical properties via classes. Extension to more complex grid-based systems can also be made based on this simplified initial test model.

Figure 2 sketches the basic operations performed by the simulation. The code includes three event handlers, one for each type of event. Much of the complexity of the event computation is encapsulated within the insert and update operations. An insert operation includes an "insert" queue operation (data structures representing particles in the cell are organized in a priority queue) and computation of the exit time of that particle from the cell based on an equation of motion. An update operation may trigger an expensive wakeup computation that again scans the particle list and updates each particle's exit time.

[1] The spacecraft situated at one end of the spatial coordinate can be treated as a special cell that does not keep the physical states of particles. We use "cells" thereafter to refer to the regular cells unless otherwise specified.

It may be noted that the reverse computation techniques introduced in [1] would generate the reverse code for each instruction without taking into account the semantics of the higher level operations that are being performed. This will clearly lead to inefficiencies for the queue management operations used in this code, and as mentioned earlier, leads to difficulties concerning the reversibility of floating point operations. The model-specific approach taken here involves generating the reverse code for the application by exploiting knowledge of the higher level semantics of the operations being performed. We call this approach *application-level* reverse computation. Here, each cell must manage a large number of particles within its domain. The choice of container class directly affects the efficiency of the simulation. We use the list class because of its efficient insertion and deletion operations. However, both insertion and deletion are destructive due to pointer assignments and thus irreversible. However, it is clear from a higher level examination of the operations being performed that insertion and deletion are perfect inverse operations of each other. Indeed, reverse computation in such queue operations are more memory-efficient than state-saving.

Careful readers may notice that the particle departure event handler depicted in figure 2 does not specify any deletion operation that matches the insertion operations in the particle arrival event handler. In fact, deletions are performed aggregately on an as-needed basis. This is done because of performance considerations. Instead of deleting each particle at its exiting time, a near-periodic deletion operation is used to amortize the cost of deletions in queues. It is observed that wakeup events happen almost periodically with a frequency determined by physical conditions within the simulation; furthermore, each wakeup requires a scan of the cell's particle queue. We find that restricting deletion operations only at cell wakeup times reduces the total overhead of particle deletions without introducing excessive memory usage. Since the aggregate deletions are used to clean up the obsolete states for particles that have already exited, no rollbacks are needed to recover these obsolete states in the event of undoing a wakeup.

Figure 3 shows the reverse code of the simulation shown in figure 2. Notice that after decomposing the forward code into components based on simulating physical processes, the reverse code is relatively easy to construct based on the operations shown in figure 3. The next step for generating the complete reverse code is just a matter of reversing each physical process. Examples of reversing some of the more difficult processes are shown in figure 3.

```
Shell::undo_arrival( ParticleArrivalEvent *e ) {
    if ( cell was activated ) {
      undo_activate cell;
    } else if ( cell already active ) {
      delete particle in cell;
      undo_update cell state;
    }
}
Shell::undo_departure( ParticleDepartureEvent *e ) {
    if ( particle was bounced from right neighbor ) {
      undo_bounce particle;
    } else {
      undo_update cell state;
    }
}
Shell::undo_inject( ParticleInjectEvent *e ) {
    delete beam particles;
    undo_update cell state;
}
```

**Figure 3:** A simplified reverse code of the PIC model

The insert operation appears in both particle arrival and injection event handlers. Its effect includes assigning memory for the new particle states in the queue and scheduling arrival/departure event pairs at each particle's future exiting time. Conversely, the delete operation in the reverse code should perform corresponding inverses of these processes. Particles in each cell are organized in a FILO (first-in last-out) queue, so the delete operation always removes the particle at the head of the queue that is exactly the same particle that was inserted in the forward computation. As for "undoing" event scheduling, it is assumed the underlying simulation engine provides the application with a primitive for explicitly retracting scheduled events; this is very useful to implement the delete operation.

The effect of the activate process is to load an inactive cell where previously no particle movements are allowed with particles of uniform distribution. Its basic operation is in fact multiple particle insertions. Its reverse code can utilize the insert-delete pair operations described above.

The cell state update process is actually the most complex process and its reverse code is not trivial. Each update performs two major computations: compute the cell's new field values and then update the cell's particle queue. Each cell computes its field locally and keeps track of field values at its left and right boundaries by summing the charges passing through that boundary. During a cell's field update, an addition of charge at its boundary can be simply reversed as a subtraction of the charge at the boundary upon rollback. The update on a particle queue, however, is not as easy. It requires a check of the wakeup condition (i.e., the field change exceeding a threshold) and

triggers a wakeup event if necessary. As previously described, a cell wakeup is the single most costly operation in the plasma simulation. In the event of a wakeup, all particles' states are recomputed based on the new cell field values and obsolete particle states are erased. The destructive nature of the recomputation is one of the principal challenges in generating the reverse code for this simulation.

One example irreversible operation is the calculation of a particle's exit time *MoveTime*. It is calculated by finding the roots *dt* of the quadratic equations [4]:

$$\frac{1}{2} * Acc * dt^2 + Vel * dt + Pos - CellWidth = 0 \quad (1)$$

$$\frac{1}{2} * Acc * dt^2 + Vel * dt + Pos = 0 \quad (2)$$

where Acc, Vel, Pos are particle acceleration, velocity and position in cell, respectively. Equations (1) and (2) represent the right and left exit conditions, respectively. dt is the time difference between the current simulation time and the particle's last movement time. The smallest real value of dt is used for MoveTime. An initial inspection of the quadratic equations seems to suggest the impossibility of applying reverse computation to this process. However, if we take the reverse computation approach at the application level, we find that the movement of the particle is highly reversible. Based on the physical laws of particle motion, the recovery of dt does not require the direct inverse of the quadratic equations. Indeed, as illustrated in figure 4, the particle's acceleration, velocity and position states can be simply rolled back by reverse computation and then the critical state dt can be reconstructed by carrying the forward computation using the recovered particle states. Please note that parameters cell_field and dt in the reverse code refer to the rolled-back cell_field value and the time difference between now and the time when the particle moved right before the wakeup. Finally, we note that random number generation is essential to this parallel simulation. Therefore, an efficient random number generator (RNG) that is reversible and has a long period is required. For this purpose, we use the reversible RNG that is described in [1].

## 5. Performance Evaluation

The application-level reverse computation approach is best implemented in a system that decouples implementation details of the simulation engine from the application in order to allow one to focus one's efforts on application semantics. As a result, a simulation engine that can support reverse computation at an application level and provide efficient management of large numbers of events with minimal storage requirement is needed. In addition, the simulation engine should have the flexibility and extensibility to support future refinement of the parallel simulation.

```
Particle::update_position( double dt ) {
    Pos += Vel * dt + 0.5 * Acc * dt * dt;
}
Particle::update_velocity( double dt ) {
    Vel += Acc * dt;
}
Particle::update_acceleration( double cell_field ) {
    Acc = cell_field * particle_charge / particle_mass;
}
Particle::reverse_position( double dt ) {
    Pos = Pos - Vel * dt - 0.5 * Acc * dt * dt;
}
Particle::update_state( double cell_field, double dt ) {
    update_position(dt);
    update_velocity(dt);
    update_acceleration(cell_field);
    dt = update_dt(); // solve eq. (1) and (2)
    MoveTime = now + dt;
}
Particle::reverse_state( double cell_field, double dt ) {
    update_acceleration(cell_field);
    update_velocity( -dt );
    reverse_position(dt);
    dt = update_dt();
    MoveTime = now + dt;
}
```

**Figure 4:** The reverse code example of the particle states

The parallel simulation code using reverse execution described in the previous section was implemented using μsik, a general-purpose parallel/distributed simulation engine based on a micro-kernel architecture [11]. μsik provides primitives supporting multiple synchronization approaches, including optimistic and conservative synchronization, as well as means to relax event ordering rules and mixing different approaches to synchronization within a single parallel execution. It therefore provides the capabilities needed for the parallel physical system simulations described here.

In a μsik simulation, logical (simulation) processes (LP) are fully autonomous entities that communicate via events. In our simulation model, each cell is implemented as an LP and can choose to run conservatively or optimistically. The conservative implementation of the simulation described here performed very poorly, due to poor lookahead, and is not discussed further. We focus on optimistic execution using our reverse handlers to support rollback.

### 5.1. Experiment Configuration

To demonstrate the feasibility and efficiency of reverse computation in the electrostatic plasma simulation, we

carried out all experiments on a Symmetric Multi-Processor (SMP) machine running Red Hat Linux 7.3 with a customized 2.4.18-10smp kernel. The SMP machine is equipped with eight Pentium III 550MHz Xeon processors that share 4GB of memory.

We use normalized units throughout our simulation, where length, time and velocity are normalized to electron Debye length, electron plasma frequency and electron thermal velocity, respectively. The spacecraft is assumed to have 500 units in radius and each cell has a width of 0.24. The solar wind plasma is initially loaded with uniformly distributed electrons and protons. We choose the initial values of 30 electrons and 30 protons per cell. The injected positron beam has energy of 10 kev with an injection period of 0.004. Upon initialization, there are up to 7000 cells of which the first 70 close to the spacecraft are active; as the simulation progresses up to time 60, the beam travels further away from spacecraft surface and thus more cells are activated.

## 5.2. Parallel Performance

Figure 5 shows a snapshot visualization of phase space structures for the solar wind electrons, protons and beam particles from a time-stepped simulation and the optimistic PDES simulation. Both simulations are run up to 60 time units, and with the same simulation parameters except using different random number generators (RNG). The PDES used a specialized reversible RNG in contrast to the generic single-stream RNG used in the time-stepped simulation. Despite this difference, the two phase space structures at the end of the simulations are rather close in form. The result from the PDES execution with reverse computation is verified to accurately capture the main features of movement for all three species of particles at the end of the simulation. In particular, we can see that the beam front in both simulations has propagated to the same distance and beam particles display a similar shape in phase space. It is also evident that the electron phase space has a finer resolution in the PDES case for up to 4000 cells. This is the result of its fine time-scale based on individual particles. A minor difference to note is that, in the PDES case, the phase space does not extend all the way to the right wall, whereas in the time-stepped model, it does. This is because we model an expanding box in PDES but not in time-stepped model. Overall, the results helped serve as validation of our optimistic simulation model against the original sequential simulation model.

The speedup of DES over TDS has been discussed in great detail in [4]. In our work, we focus on further improving the DES performance by realizing parallelization in the simulation and utilizing optimistic synchronization. All the parallel experiments discussed in

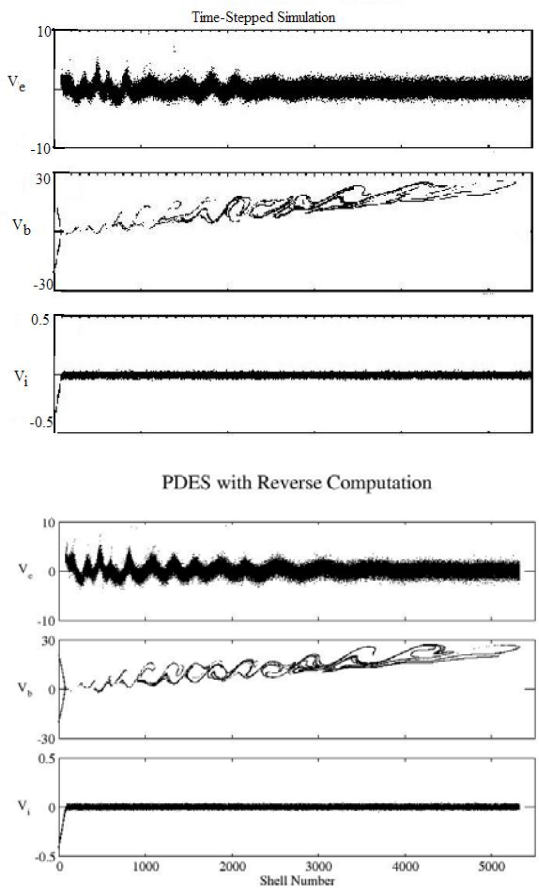


**Figure 5:** Validation by phase space comparison of time-stepped simulation and PDES with reverse computation.

the following section were run with the same physical parameters and resulted in the same number of committed events as the sequential runs.

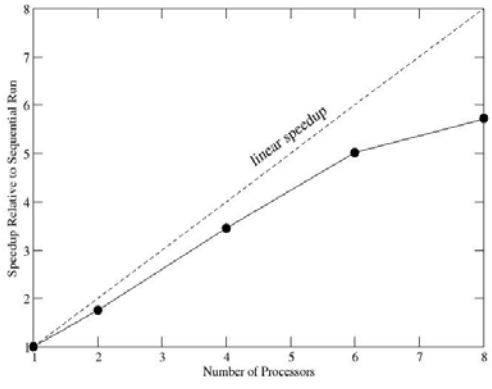


**Figure 6:** PDES vs. sequential DES

Figure 6 shows the parallel speedup in terms of

execution time for up to 8 processors. The sequential data is measured by running the parallel code on a single processor. It should be noted that the single processor execution incurs neither rollbacks nor state saving overhead. Because μsik was designed for both efficient sequential and parallel execution, we believe these measurements reflect the performance one could expect to see using a reasonably efficient sequential simulation engine.

We observe that the optimistic parallel execution achieves a nearly linear speedup up to 4 processors, but the performance improvement is somewhat less in going from 4 to 8 processors. This phenomenon is largely due to the fact that there is relatively little computation per particle event. As the computation is distributed over more and more processors, the amount of computation between event communications decreases, resulting in reduced speedup. We expect that this problem will not persist if a larger, more complex physical model such as a three dimensional plasma code were used. An initial test with an increased simulation time of 2 units did show better speedup performance due to the fact that the longer the simulation runs, the more cells are activated, resulting in more balanced computation for each processor.

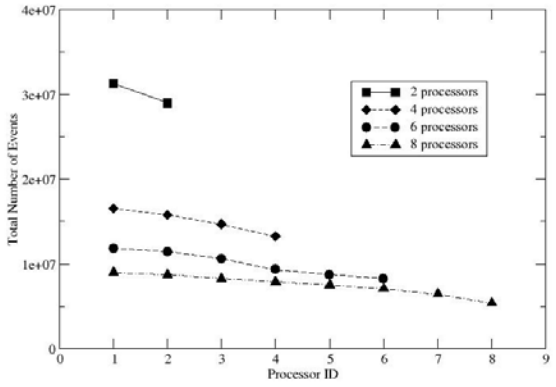


**Figure 7:** Event rate distribution

A second factor that results in less than optimal performance concerns the distribution of the workload. Figure 7 shows the amount of computation assigned to each processor in each of the runs. Here, the load is distributed by first dividing the physical area encompassed by the simulation into two regions (as illustrated in figure 1) with the initially active cells closer to the spacecraft in the "heavy activity" region, and other cells forming the "less active" region. Cells in the active regions are evenly grouped and distributed among the available processors, while other cells are grouped into sub-regions or "blocks" and distributed among processors in a round-robin fashion. All simulations shown in figure 7 have a fixed "block" size. We observe that during the lifetime of each simulation, the processor load shows significant variation as more and more cells become active. Upon simulation termination, the simulation with the largest number of processors tends to be the least balanced. The imbalance is inherent of such simulations due to their highly dynamic nature and the static load-balancing scheme. Further investigation of characteristics of electrostatic plasma simulations is needed to aid in the development of a more efficient load-balancing algorithm for this application that can lead to better parallel performance for large numbers of processors.

### 5.3. Efficiency

Intuitively, grid-based physical systems such as the electrostatic plasma simulation studied here have the desirable features of locally solved field values and queuing/dequeuing operations that are time-reversible, but the evolution of the system itself (beam injections, cell wakeups in our case) is not time-reversible. However, with the application-level reverse computation illustrated above, we have shown that numerical operations in the electrostatic plasma simulation chosen for this study are truly reversible, despite round-off errors and irreversible evolution processes. The most important discovery from our study is that application-level reverse computation may be quite efficient for these scientific simulations.

The efficiency mainly comes from two contributing factors: the smaller amount of memory required compared to state-saving, particularly, queue operations where no additional state is required to perform rollbacks; no the fact that the simulation is not constrained by arbitrarily small look-ahead values. However, there are still important practical issues related to reverse computation.

Ideally, one would like to apply reverse computation to all reversible operations. But reverse computation also comes at a cost: if the number of destructive operations is sufficiently large and no efficient application-level reverse computation can be found, stubbornly employing reverse computation can result in worse performance than state-saving. One such case as pointed out in [1] is when a rollback spans several processed events. Merely switching pointers to restore a state based on the earliest rolled back event incurs a small cost in copy state-saving; while reverse computation must roll back one event at a time and thus excessive rollbacks can cause performance to degrade considerably. The effect of this is particularly severe in our simulation when a rollback spans multiple wakeup events.

Our solution to reducing the rollbacks of costly wakeup events is by limiting the "optimism" of the parallel processing. μsik supplies the simulation applications with a convenient facility for our purpose. A "*run-ahead*" parameter can be set by the model at simulation

initialization to limit how far in simulation time each LP can run ahead of other LPs during optimistic execution. By carefully tuning the run-ahead parameter based on cell wakeup frequency, we are able to reduce or eliminate consecutive rollbacks of wakeup events.

In addition to the basic reverse computation techniques discussed here, advanced reverse techniques can be applied to the plasma simulation. For example, compiler-supported reverse computation can be used to further optimize the parallel performance at run-time. This approach is beyond the scope of our discussion and will be studied in the future.

## 6. Conclusions

In this work, we have applied reverse execution to perform parallel discrete event simulations of a physical system. We demonstrated that application-level reverse computation can be used to manually generate efficient reverse code. These results suggest that reverse computation merits further investigation as an approach for parallel/distributed simulation of physical systems modeled using a discrete event simulation paradigm.

As previously mentioned, the PIC simulation considered in this paper is only a simplified example of reverse execution in simulating physical systems. The examples given in section 4 are representative and certainly do not encompass the diversity and complexity of all physical system simulations. However, the underlying reverse techniques can be used in other grid-based models without extensive modifications. Here we provide some guidelines for the development of parallel physical discrete event simulations using reverse computation. Since our exploration of reverse computation is an on-going research effort, the guidelines provided here should be used as a references rather than strict rules for applying reverse computation in scientific simulations.

- Reverse computation is well-suited for fine-grained applications such as the 1D electrostatic grid-based plasma models. It is especially useful where efficient queue management is needed. But other optimization techniques should also be considered in order to fully optimize parallel performance.
- Good knowledge of the application semantics, especially the underlying physics, can be beneficial in producing reverse code for physical systems. Model-specific optimization can be quite efficient but requires knowledge of application-level operations. The simple example of reversing the quadratic equation would not have been efficient, if at all possible, without knowledge of the physics involved (particle's motion in this case).
- The modeling process largely determines how successfully reverse computation will improve parallel performance. Initial analysis in [1] shows that complex use of jump instructions such as `goto`, `break` and `continue` are difficult to optimize in terms of memory usage.
- In modeling physical systems, one should attempt to avoid monolithic code for event handlers and use functions calls that are associated with each physical process. If an event handler only consists of a long sequence of simple instructions, it is difficult to exact application semantics and therefore reverse computation will degenerate to instruction-by-instruction reverse execution. Using many small function calls that reflect physical processes helps to develop reverse codes based on physical properties of the system. Another advantage is easier debugging and testing for the reverse code.

The work presented here is only an initial step based on a simplified physical system. Yet, the results show promise. Our goal is to build a scalable parallel simulator for complex physical systems by exploitation of more advanced reverse computation techniques.

## References


[1] Carothers, C. D., K. Perumalla and R. M. Fujimoto. Efficient Optimistic Parallel Simulation Using Reverse Computation. ACM Transactions on Modeling and Computer Simulation **9**(3): 224-253, 1999.

[2] Birdsall, C.K. and A.B. Langdon. Plasma Physics via Computer Simulation. McGraw-Hill Book Company, 1985.

[3] Dawson, C. and R. Kirby. High Resolution Schemes for Conservation Laws with Locally Varying Time Steps. SIAM Journal of Scientific Computing, 2001. **22**(6): p. 2256.

[4] Karimabadi, H, Driscoll, J, Omelchenko, Y.A. and N. Omidi. A New Asynchronous Methodology for Modeling of Physical Systems: Breaking the Curse of Courant Condition. J. Computational Physics, 2004, submitted.

[5] Hontalas, P., B. Beckman, M. DiLorento, L. Blume, P. Reiher, K. Sturdevant, L. V. Warren, J. Wedel, F. Wieland and D. Jefferson. Performance of the Colliding Pucks Simulation on the Time Warp Operating System. Distributed Simulation, Society for Computer Simulation International, 1989.

[6] Lubachevsky, B. D. Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks. Communications of the ACM **32**(1): 111-123, 1989.

[7] Lubachevsky, B. D. Several Unsolved Problems in Large-Scale Discrete Event Simulations. Workshop on Parallel and Distributed Simulation, 1993.

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

1985.
[9] Yuan, G., C. D. Carothers and S. Kalyanaraman. Large-Scale TCP Models Using Optimistic Parallel Simulation. Workshop on Parallel and Distributed Simulation, 2003.
[10] Pritchett, P. L., R. M. Winglee. The Plasma Environment during particle beam injection into space plasmas: 1. Electron-beams, J. of Geophysical Res. – Space Physics, 92 (A7): 7673-7688 JUL 1 1987.
[11] Perumalla, K. μsik - A Micro-kernel for Parallel/Distributed Simulation. Technical Report GIT-CERCS-04-20, Center for Experimental Research in Computer Science, Georgia Institute of Technology, 2004.