Parallel Discrete Event Simulations of Grid-based Models: Asynchronous Electromagnetic Hybrid Code

H. Karimabadi[1], J. Driscoll[1], J. Dave[1,2], Y. Omelchenko[1],K. Perumalla[2],R. Fujimoto[2], N. Omidi[1]
[1]SciberNet, Inc., Solana Beach, CA, 92075, USA
{homak, driscoll,yurio,omidi}@scibernet.com
[2]Georgia Institute of Technology, Atlanta, GA, 30332, USA
{jagrut,fujimoto, kalyan}@cc.gatech.edu

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. Recently, we proposed [1] a new asynchronous formulation 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. Using a serial electrostatic implementation, we obtained more than two orders of magnitude speed up compared with traditional techniques. Here we examine the issues related to the parallel extension of this technique and discuss several different parallel strategies. In particular, we present in some detail a newly developed discrete-event based parallel electromagnetic hybrid code and its performance when run in the conservative mode on a cluster computer. These initial performance results are encouraging in that they demonstrate very good parallel speedup for large-scale simulation computations containing tens of thousands of cells, though overheads for interprocessor communications remain a challenge for smaller computations.

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 [2] 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 yet exist.

We have taken a new approach [1] to the simulation of such complex systems. The conventional time-stepped grid-based particle-in-cell (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. The technique has some similarity to cellular automata (CA) in that complex behaviors result from interaction of adjacent cells [3]. However, unlike CA, the interactions between cells are governed by a full set of partial differential equations rather than the simple rules as are typically used in CA. Furthermore, the power of the technique is in its asynchronous nature as well as elimination of unnecessary computations in regions where there is no significant change in time. This is in contrast to CA which are largely based on synchronous computation models (e.g., [4]); to date, asynchronous parallel discrete event simulation of CA have only been applied to relatively simple phenomena such as Ising spin [5]. Event-driven PIC simulations automatically guarantee that the progression of the system evolution over time captures important state changes without processing "idle" information.

Using an electrostatic and serial model, we have shown [1] that the discrete event technique leads to more than two orders of magnitude speed up compared to conventional techniques. In the following, we discuss the issues associated with the extension of the technique to parallel architecture. We then demonstrate, through a newly developed parallel hybrid code, that parallel processing can provide an additional order of magnitude improvement in performance.

2 Parallel Computation Issues

Because time steps in parallel discrete event simulations (PDES) are not uniform, alternative techniques and metrics must be considered. For example, the irregular nature of PDES computations leads to difficult problems concerning synchronization. This issue has received much attention in the PDES literature [6]. Synchronization overhead, number of concurrent simulation computations, load distribution, and the rate of event processing are all significant factors impacting PDES performance.

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 that 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.

By 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 intervals 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) [7]. 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 ("lookahead") 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 must be performed. While exploitation of discrete event simulation techniques will provide dramatic performance improvements, by itself, DES is not sufficient to achieve the desired performance and scalability for accurately simulating massive and complex physical systems like the earth’s magnetosphere using realistic parameters.

- **Synchronization**: This is by far the paramount issue to be carefully resolved for achieving the best parallel execution performance. Broadly there are four approaches commonly used: conservative, optimistic, relaxed, and combined synchronization. In the following, we assume the parallel simulation computation is composed of a collection of simulation processes that communicate by exchanging time stamped event messages.
    - **Conservative**: This approach always ensures *safe* timestamp-ordered processing of simulation events within each process [8, 9]. In other words, a simulation process does not execute an event until it can guarantee that no event with a smaller timestamp will later be received by that process. 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 interactions with other processes 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 easier to implement than the optimistic approach (described next) if the lookahead can be specified by the application [6].
    - **Optimistic**: This approach avoids blocked waiting by optimistically processing the events beyond the lookahead window [7]. 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. A key issue introduced by large-scale platforms is the increased delay of inter-processor communication. Optimistic synchronization offers the potential for greater resilience to delays in the sense that computations may progress despite the absence of certain data. 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. Traditional optimistic methods rely on state saving (incremental or copy) to enable rolling back to a previously saved state, in case an event arrives in the "past". This is problematic for the large-scale and complex simulation models discussed here due to memory and computation constraints. Many events contain relatively little computation (a few microseconds each), so small overheads would result in very significant performance degradations. Use of reverse execution techniques rather than state saving offer one approach to addressing this concern.

    - **Relaxed synchronization:** This approach relaxes the constraint that events be strictly processed in time stamp order (e.g., see 10-11]). For example, it might be deemed acceptable to process two events out of order if their time stamps are "close enough." This approach offers the potential of providing a simplified approach to synchronization, but without the lookahead constraints that plague conservative execution. A key challenge with this approach is determining the extent that ordering constraints can be relaxed without compromising the validity of the simulation. An additional challenge lies in ensuring that the execution of the simulation is repeatable. Repeatability means multiple executions of the same simulation with the same inputs are guaranteed to yield the same numeric results from one execution to the next. This property may not be preserved because events within each process may be processed in a different order from one execution to the next unless care is taken to ensure that this is always the case.
    - **Combined synchronization**: This approach combines elements of the previous three. For example, 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) (e.g., see 12). In such cases, a combination of 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 and interprocessor communication is minimized (often these are conflicting goals). In parallel discrete event 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. Further, if optimistic execution is used, one must be able to take into account rolled back computation, as it has the same effect as an idle processor.
    - **Automated/Adaptive**: Automated schemes are preferable for load-balancing at runtime. These schemes vary with the particular synchronization approach used. For example, using process migration to balance workloads presents new challenges in this area [13-15].
    - **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.

2.1 Musik

The need for a simulation engine which is easily extensible to provide a variety of synchronization mechanisms and event delivery mechanisms through a single interface prompted the development of Musik [16]. The modeling and runtime interface is also kept abstract and flexible, so that radically alternative implementations can be implemented underneath the interface. It is analogous to the microkernel of an operating system. The responsibility of a micro-kernel is restricted to only providing core services; in this case ensuring that the simulation processes can efficiently communicate with each other, and collectively accomplish time-ordered processing of millions of events. Musik provides primitives supporting various types of optimistic, conservative, relaxed, and combined synchronization. While using a common engine to support multiple synchronization protocols is not new (e.g., see 17), the parsimonious API and the high-performance implementation approach is novel.

A Musik simulation consists of different *Simulation Processes* that communicate via events. Processes may be on the same processor or on a remote processors connected by a network. Each process is an independent entity, and can choose to be conservative or optimistic. The simulation engine guarantees the overall correctness of the simulation. It includes a facility to track event-related timing, as well as non-event overheads, which mostly consist of communication overheads, at run-time. Figure 1 depicts a simulation using Musik. SP stands for a simulation process, also referred to as logical process.

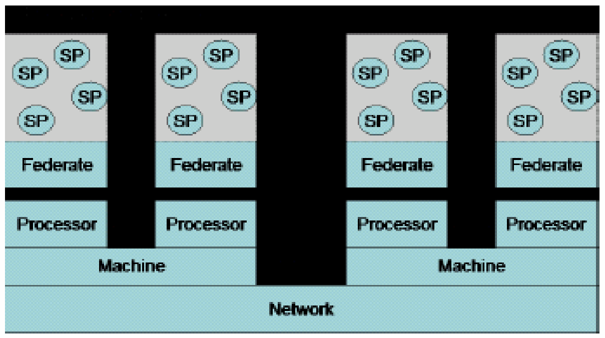

Fig. 1. A Musik Simulation

3 DES Model

We have developed a general architecture for parallel discrete event modeling of grid-based models. We have also developed models codes for the diffusion problem, electrostatic code, and electromagnetic hybrid code. Each of these model codes come in two versions, a "light" version and a "pro" version. The light versions have simpler algorithms and are used for the testing of various concepts that are then incorporated into the pro versions. The discussion of the general architecture and its many subtleties (e.g., flux matching algorithm, our new parallel scheme called "preemptive event processing", etc.) are beyond the scope of this article and will be presented elsewhere. Here we present a simplified version of our technique that illustrates the salient feature of our model without getting bogged down in all the details. In the following, we show results from a 1D parallel hybrid code (light version) that we have developed and tested using Musik as the simulation engine. The light version does not strictly conserve flux, nor does it eliminate all the unnecessary computation in the inactive regions. As we will demonstrate shortly, for the shock problem of interest here, lack of strict flux conservation does not change the result significantly. Furthermore, the code is very useful in highlighting the unique parallel issues that are encountered in the DES modeling of plasmas. Details of the pro versions of these codes will be presented elsewhere.

3.1 Hybrid Algorithm

In hybrid codes, ions are treated as macroparticles whereas electrons are treated as a fluid (electron moments up to and including temperature are retained). Hybrid codes are ideally suited for physical phenomena that occur on ion time and spatial scales and where a kinetic description of the electrons is not required. Maxwell's equations are solved by neglecting the displacement current in Ampere's law (Darwin approximation) that eliminates light waves, and by explicitly assuming charge neutrality. There are several variations of the electromagnetic hybrid (fluid electrons, kinetic ions) algorithms [18]. Here we use the one-dimensional resistive formulation [19] which casts the field equations in terms of the vector potential. The reader is referred to [19] for details regarding the field equations and their finite difference forms that are solved here. The model problem uses the piston method where incoming plasma moving with flow speed larger than its thermal speed is reflected off the piston located on the rightmost boundary. This leads to the generation of a shockwave that propagates to the left. In this example, we use a flow speed large enough to form a fast magnetosonic shock. In all the runs shown here, the plasma is injected with a velocity of 1.0

(normalized to upstream Alfven speed), the background magnetic field is tilted at an angle of 30^o, and the ion and electron betas are set to 0.1.

The simulation domain is divided into cells [1], and the ions are uniformly loaded into each cell. Here, we conducted experiments ranging from 4096 to 65,536 cells, and initialized each simulation to have 100 ions per cell. Each cell is modeled as a *Simulation Process* in Musik, and the state of each SP includes the cell's field variables. The main tasks in the simulation are: Initialize fields, Initialize particles, Calculate the exit time of each particle, Sort IonQ, Push particle, Update fields, Recalculate exit time, Reschedule. This is accomplished through a combination of priority queues and three main classes of events. The ions are stored in either one of two priority queues as illustrated in Figure 2. Ions are initialized within cells in an *IonQ*. As ions move out of the left most cell, new ions are injected into that cell in order to keep the flux of incoming ions fixed at the left boundary. The placement/removal of the ions in *IonQ* and PendQ are controlled by their MoveTime (time at which an ion is to be moved next) compared to the current time and lookahead (see below). IonQ is sorted by MoveTimes. Ions which have MoveTimes more than Current Time + 2*lookahead have not yet been scheduled and are kept in the IonQ. On a wakeup, only the ions in this queue recalculate their MoveTimes. Because ions in the IonQ have not yet been scheduled, a wakeup requires no event retractions. If an ion's MoveTime becomes less than Current Time + 2*lookahead in the future, the ion is scheduled to move, and is removed from the IonQ and placed in the PendQ. Thus the front of the IonQ is always at least one lookahead period ahead of the current time. This guarantees that each ion move will be scheduled at least one lookahead period in advance. The PendQ is used to keep track of ions that have already been scheduled to exit, but have not yet left the cell. These particles have MoveTimes that are less than the current time. Any ions kept in the PendQ which have MoveTimes earlier than the current time have already left the cell and must be removed before cell values such as density and temperature are calculated.

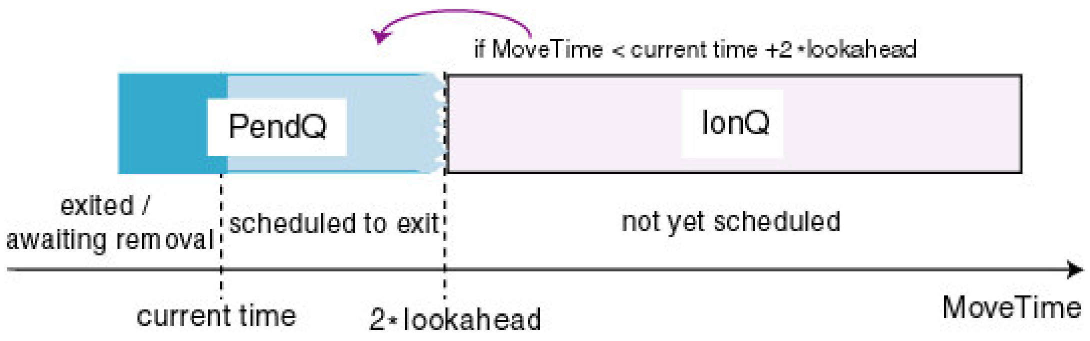

Fig. 2. At any given time, the ions in a given cell are stored in one of two queues, the IonQ and the PendQ. Both are priority queues, sorted so that the ion with the earliest exit time is at the top.

Events can happen at any simulation time, and are managed separately by individual cells of the simulation. The flow of the program including functions of events and their interaction with Musik are illustrated in Figure 3. In this simulation, each cell handles three different types of events:

SendIon Event: This event is first run on each cell when the simulation is initialized, and is responsible for sending ions from one cell to the next. This is accomplished by scheduling the complementary "AddIon" events for neighboring cells. The SendIon event will schedule an AddIon event corresponding to every Ion which will exit within 2 lookahead periods, and will always schedule at least one SendIon event. In addition, the SendIon Event checks to see if the fields have changed by some tolerance, waking up particles in that cell if necessary. The SendIon events occur frequently and as a whole are computationally significant.

AddIon Event – this event is used to add a single ion back into a cell. The ion's new exit time is calculated, and then it is added to the IonQ. The fields in the cell are then updated, and Notify Events are scheduled for the two neighboring cells to inform them of the change in the neighboring cell. The AddIon Event causes state changes and can occur sporadically in large batches.

NotifyEvent – updates the register vector potential A-field and temperature for the two neighbors.

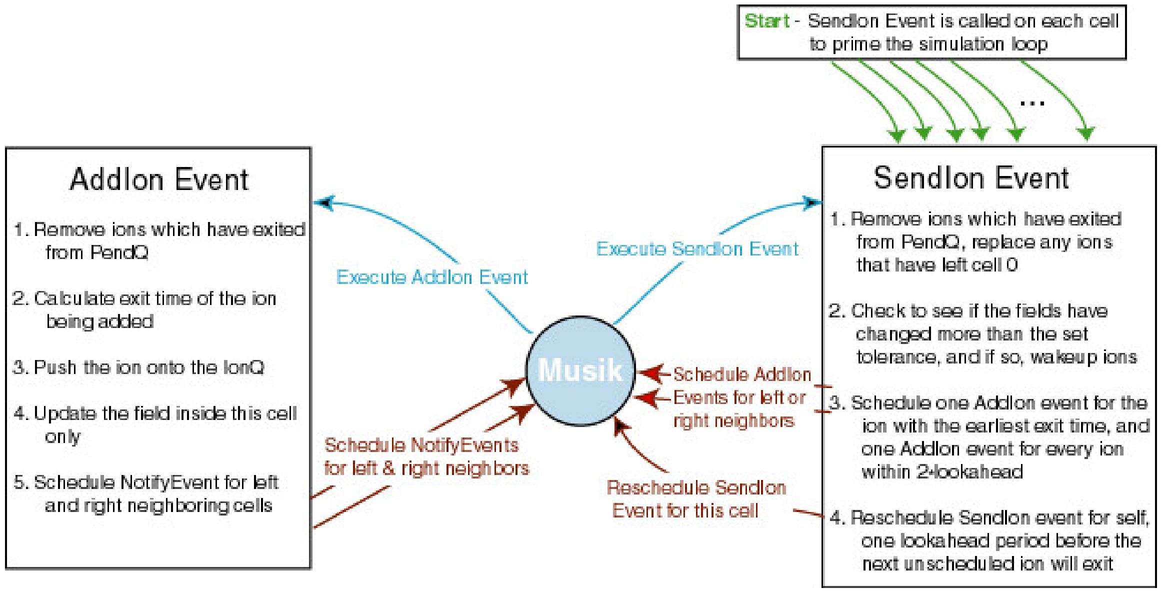
Start - SendIon Event is called on each cell to prime the simulation loop
...
AddIon Event
1. Remove ions which have exited from PendQ
2. Calculate exit time of the ion being added
3. Push the ion onto the IonQ
4. Update the field inside this cell only
5. Schedule NotifyEvent for left and right neighboring cells
Execute AddIon Event
Execute SendIon Event
Musik
Schedule NotifyEvents for left & right neighbors
Schedule AddIon Events for left or right neighbors
Reschedule SendIon Event for this cell
SendIon Event
1. Remove ions which have exited from PendQ, replace any ions that have left cell 0
2. Check to see if the fields have changed more than the set tolerance, and if so, wakeup ions
3. Schedule one AddIon event for the ion with the earliest exit time, and one AddIon event for every ion within 2•lookahead
4. Reschedule SendIon event for self, one lookahead period before the next unscheduled ion will exit

Fig. 3. Flow diagram of the parallel hybrid code.

3.1.a Exit Time

We take the electric and magnetic fields to be constant within a cell, with arbitrary orientation and magnitude. In this case, a charged particle will have an equation of motion that can be calculated analytically and has the general form $\mathbf{R}(t) = \mathbf{A}t^2 + \mathbf{B}t + \mathbf{r}_c \sin(\omega_c t + \vec{\phi}) + \mathbf{C}$, where $\mathbf{R}(\mathrm{t})$ is the position of the particle. Newton's method is used to solve for the exact exit time.

3.1.b Look ahead

In general, lookahead is an application specific parameter. If the typical velocity of a particle is *v*, and a typical cell width is *x*, then the typical time it takes for a particle to cross a cell is *x*/*v*. Obviously, lookahead must be some factor smaller than this, because you only want to allow a particle to move a small fraction of a cell distance in one lookahead period. On the other hand, if the lookahead is too small, the parallel performance will be poor. This happens when too few computations are made during one lookahead period, and the processor synchronization overheads become large compared to the computational overheads. As with any conservative parallel discrete event simulator, in Musik, an event can never be scheduled less than lookahead in the future. So lookahead can never be greater that the earliest exit time for all cells at time zero. We use this number (the time it takes for the first particle to exit a cell) to set the lookahead.

4 Results

Figure 4 shows the comparison of the results between the traditional time-stepped hybrid simulation and our event-stepped simulation using a single processor. 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 wave 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 wave speed, and (ii) details of the wavetrain associated with the shock wave. This match is impressive considering the fact that the differences seen in Figure 4 are within statistical fluctuations associated with changes in the noise level in hybrid codes.

4.1 Effect of Lookahead

Next, we consider the effects of changing the lookahead on both the accuracy of the results as well as the execution time. The hardware for the runs shown here is a high-performance cluster at the Georgia Tech HPC laboratory. The cluster has 8 nodes, each with 4 2.8 GHz Xeon processors and 4 GB of RAM. The simulation uses 4 Federates (one per processor), each with 2 Regions and 512 cells per Region. Figure 5 shows the variations in the solution, as illustrated in the spatial profile of B_{tot}, with lookahead. The zero lookahead run yields the most accurate results and is treated as a baseline. The maximum value of lookahead (~0.15) is determined by the requirement that it does not violate the property of time-ordered processing in conservative synchronizations. In the hybrid algorithm, the maximum lookahead must be less than the exit time of the earliest scheduled particle that is approximately 0.15 for our choice of parameters. When lookahead exceeds this value, the simulation schedules an event which violates the lookahead constraint imposed by conservative synchronization (scheduling an event for a cell in its "past") and cannot proceed further. The deviations from the zero look ahead are less than 10% using the maximum lookahead value.

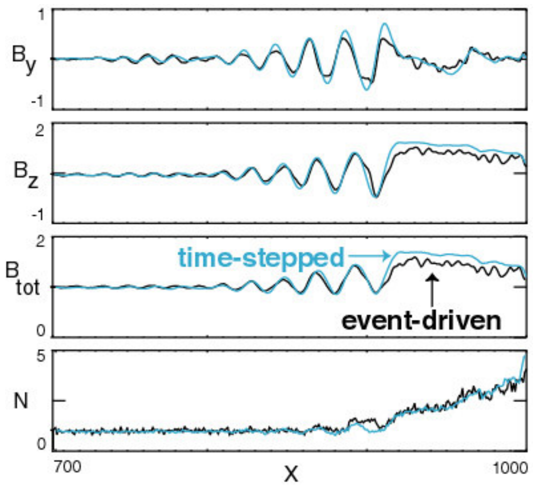

Fig. 4. Comparison of time-stepped and event-driven simulation of a fast magnetosonic shock.

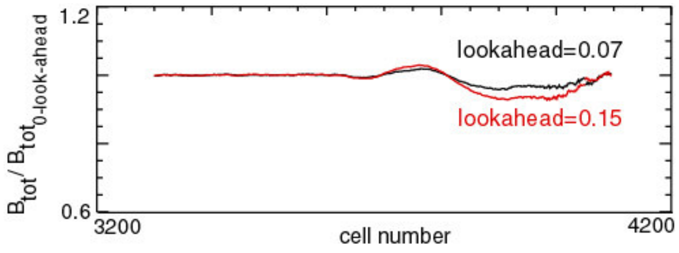

Fig. 5. The ratio of the total magnetic field from lookahead runs of 0.07 and 0.15 relative to the solution from zero lookahead.

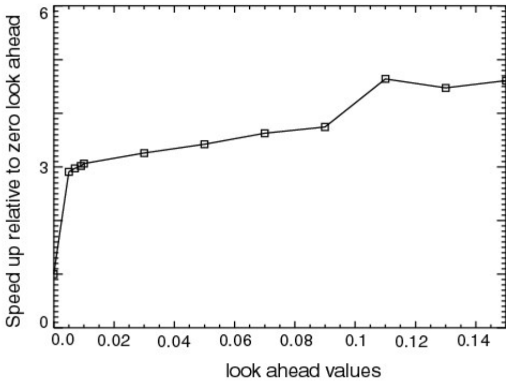

Fig. 6. Speedup with increased lookahead.

Figure 6 shows the speed up in execution time relative to the zero lookahead run. The important point from this figure is that even small departures from 0 lookahead lead to substantial increases in speed. In fact the most dramatic speedup (a factor of 3) is achieved when lookahead is changed from 0 to 0.005. Further changes in lookahead do improve performance, but at a much slower rate. For example, increasing the lookahead by an order of magnitude from 0.005 to 0.05 leads to only a 15% increase in speedup.

4.2 Scaling with the number of processors

The 1D modeling of the shock problem in Fig. 4 can be easily performed with a serial version of our code. However, our ultimate goal is to develop a 3D version of the code. As a simple means to evaluating the parallel execution in a 3D model, we have considered cell numbers as large as 65536. This is sufficient to identify the key issues in the parallel execution. Figure 7 shows the speedup as a function of the number of processors up to 128. The speedup is measured with respect to a sequential run. These runs were made on a 17 node cluster, with each node having 8 550 MHz CPUs and 4 GB of RAM. The simulation domain consists of 8192 cells in one case and 65536 cells in the other.

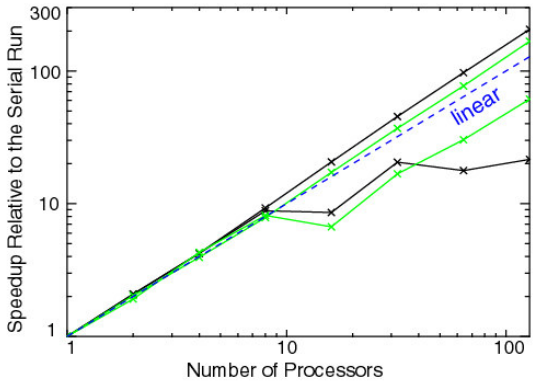

Fig.7. Scaling with the number of processors. The dashed line is a linear scaling curve. The results from two runs with simulation domain consisting of 8192 (in black) and 65536 (in green) cells are shown. In each case, the scaling is seen to exhibit superlinear scaling if communication and waiting time are taken out.

As evident from figure 7, the parallel speedup is good up through eight processors, but then declines as more processors are added. This is due to the architecture of the cluster, which uses a collection of 8-cpu computers communicating through TCP/IP. With up to eight processors, the entire simulation runs through shared memory, and the communication overheads are low. However, with more than eight processors, the overheads associated with TCP/IP begin to offset the speed gained from using more processors. This causes a reduction in the slope of the curve. For 8192 cells, the speedup does not increase significantly after 32 processors. This is because of increased synchronization cost that negate the gains from parallel processing. Processors do not have sufficient computation between global synchronizations and spend a greater fraction of time waiting on other processors. For 65536 cells, there is enough computation between global synchronizations to obtain good speedup up to 128 processors. Figure 8 shows the percentage of time spent in communication and blocking in each case. There is a significant increase in the fraction of time spent in communication time after 8 processors. For 65536 cells, the percentage of communication time settles to around 60% for higher number of processors. However, for 8192 cells, the percentage keeps on increasing until 90% for 128 processors.

Since the overheads associated with inter-processor communication become relatively smaller as the simulation size increases, we do not anticipate this effect being as pronounced with larger 3d simulations. In 3d simulations, each processor would have several orders of magnitude more cells, making the relative overheads associated with TCP/IP much less. To test the idea that poor scaling is the result of the communication overheads, we have also plotted in Fig. 7 the scaling with the communication time subtracted out. The speedup in this case is better than expected, and is in fact super-linear. In other words, doubling the number of processors more than doubles the execution speed. This is the result of a peculiar feature of DES, which is that execution time does not necessarily scale linearly with the simulation size, even on one processor. This is in part due to the fact that as the event queue becomes larger (that is, contains more pending events), the time associated with scheduling and retrieving each event increases. So as the simulation gets distributed over more and more processors, each processor is effectively dealing with a smaller piece of the simulation, making the scaling non-linear. Memory performance (specifically, cache performance) can also lead to superlinear speedup. By keeping the same size problem but distributing it over more processors, the memory footprint in each processor shrinks. This leads to better cache performance in each processor, which can have quite a substantial impact on performance. Stated differently, the total amount of cache memory increases in proportion with the number of processors used - with enough processors, one can, for example, at some point be able to fit the entire computation into the processors' caches. An extreme case of this is when the problem is so large that it does not fit into the memory of a single machine, causing paging to occur, yielding very poor performance in the sequential execution, and inflated speedup numbers with multiple processors. Although both effects could be causing the superlinear scaling seen in figure 7, the data structure performance appears to be the dominant cause in this case. In a no-load test of Musik, we changed the number of cells from ten to a million. The change in speed that Musik handled *each* event increased from 6.50 to 22.15 microseconds, following a NlogN behavior.

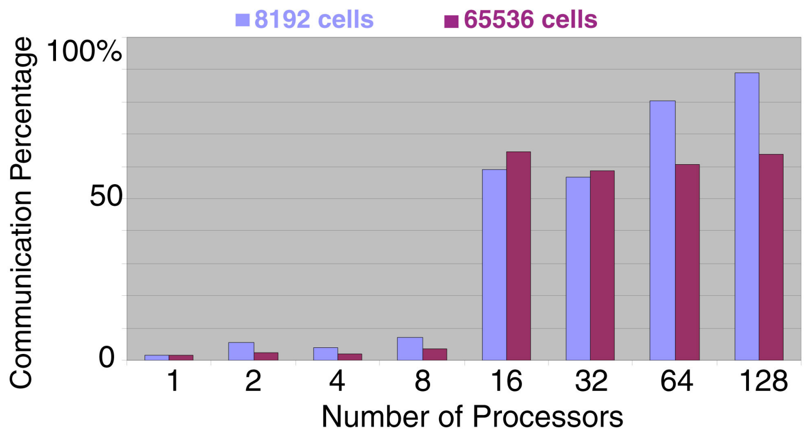

Fig.8. Percentage of time spent in communication.

4.3 Load Balancing

Figure 9 shows the variation in execution time as a function of the number of Regions per processor, as distributed by the Region Deal algorithm. In this scheme, the simulation is broken into small regions which are then "dealt" out much like a ca rd game among processors. These simulation runs were performed on the first cluster mentioned earlier.

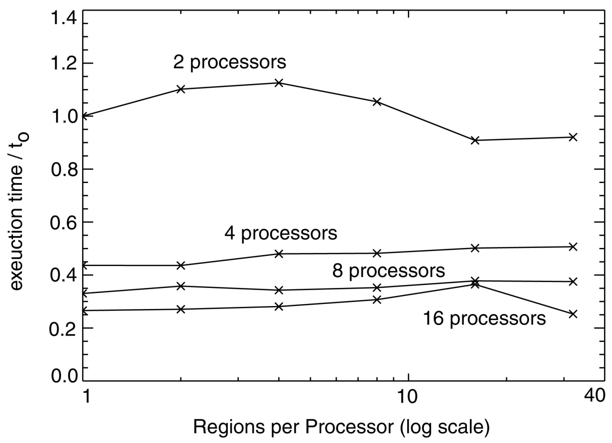

Fig. 9. Performance of load balancing algorithm.

The curves show a different trend for higher number of processors (4,8,16) than for 2 processors. For higher number of processors, the variation in execution time because of the Region Deal load balancing scheme is less pronounced. The best execution times are close to the execution time for 1 Region per processor, with variations of less than 1,000 seconds. Also, increasing the number of Regions per processor increases the execution time in most cases. This is because of the increased synchronization overhead that negates the benefits of the load distribution scheme. For 2 processors, having more Regions per processor leads to better load distribution and hence reduced execution time. The execution time settles around 14,000 seconds for 16 Regions or more. In this case too, contiguous cells are assigned to different processors and incur greater synchronization overhead for higher number of Regions per processor.

Acknowledgements This research was supported by the National Science Foundation Information Technology Research (ITR) grant numbers 0325046 at SciberNet, Inc. and 0326431at Georgia Institute of Technology. Some of the computations were performed at the San Diego Supercomputer Center.

References

1. 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.
2. Karimabadi, H. and N. Omidi. *Latest Advances in Hybrid Codes and their Application to Global Magnetospheric Simulations*. in *GEM, http://www-ssc.igpp.ucla.edu/gem/tutorial/index.html)* (2002).
3. Ilachinski, A., Cellular Automata, A Discrete Universe, World Scientific, 2002.
4. Smith, L., R. Beckman, et al. (1995). TRANSIMS: Transportation Analysis and Simulation System. Proceedings of the Fifth National Conference on Transportation Planning Methods. Seattle, Washington, Transportation Research Board.
5. Lubachevsky, B. D. (1989). "Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks." Communications of the ACM **32**(1): 111-123.
6. Fujimoto, R.M., *Parallel and Distributed Simulation Systems*. (2000): Wiley Interscience.
7. Jefferson, D., *Virtual Time,* ACM Transactions on Programming Languages and Systems, (1985), 7(3):pp. 404-425.
8. Chandy, K. and J. Misra (1979). Distributed Simulation: A case study in design and verification of distributed programs. IEEE Transactions on Software Engineering.
9. Chandy, K. and J. Misra (1981). Asynchronous distributed simulation via a sequence of parallel computations. Communications of the ACM. **24.**
10. Fujimoto, R. M. (1999), Exploiting Temporal Uncertainty in Parallel and Distributed Simulations, Proceedings of the 13th Workshop on Parallel and Distributed Simulation: 46-53.
11. Rao, D. M., N. V. Thondugulam, et al. (1998). Unsynchronized Parallel Discrete Event Simulation. Proceedings of the Winter Simulation Conference**:** 1563-1570.
12. Rajaei, H., R. Ayani, et al. (1993). The Local Time Warp Approach to Parallel Simulation. Proceedings of the 7th Workshop on Parallel and Distributed Simulation**:** 119-126.
13. Boukerche, A., and S. K. Das, "Dynamic Load Balancing Strategiesfor Conservative Parallel Simulations," Workshop on Parallel and Distributed Simulation, 1997.
14. Carothers, C. D., and R. M. Fujimoto, "Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 11, No. 3, pp. 299-317, March 2000.
15. Gan, B. P., *et al.*, "Load balancing for conservative simulation on shared memory multiprocessor systems," Workshop on Parallel and Distributed Simulation, 2000
16. Perumalla, K.M, *μsik – A Micro-Kernel for Parallel/Distributed Simulation*, Georgia Tech Technical Report GIT-CERCS-TR-04-20, http://www.cercs.gatech.edu/tech-reports/tr2004/git-cercs-04-20.pdf.
17. Bagrodia, R., R. Meyer, et al. (1998). "Parsec: A Parallel Simulation Environment for Complex Systems." IEEE Computer **31**(10): 77-85.
18. Karimabadi, H., D. Krauss-Varban, J. Huba, and H. X. Vu, On magnetic reconnection regimes and associated three-dimensional asymmetries: Hybrid, Hall-less hybrid, and Hall-MHD simulations, J. Geophys. Res., (2004), in press.
19. Winske, D. and N. Omidi, *Hybrid codes: Methods and Applications*, in *Computer Space Plasma Physics: Simulation Techniques and Software*, H. Matsumoto and Y. Omura, Editors. (1993), Terra Scientific Publishing Company. p. 103-160.