RELAXATION FOR MASSIVELY PARALLEL DISCRETE EVENT SIMULATION

Boris D. Lubachevsky

AT&T Bell Laboratories
Murray Hill, NJ 07974, USA

ABSTRACT

The discussion in this tutorial is centered around a new space-time relaxation paradigm which appears to be a good candidate for rendering efficient a wide class of massively parallel discrete event simulations.

1 Discrete event simulation: what it is and why we consider it

Several forms of computer rendering, such as computer graphics or computer games, are sometimes called "simulations." Here, we focus attention on simulating a *dynamic system*. Simulation will be understood to be the process of generating the trajectory or sample path for such a system.

Typically, the user "pushes the limits" and simulates as large a model as computer memory permits and for as long as it is tolerable. Hence it is not surprising that a substantial fraction of the non-idle CPU time of many computers, perhaps, as large as 50%, is occupied with simulations. It might seem somewhat unexpected, though, to find a substantial fraction of all simulations to be dealing with *discrete event* models.

After all, most of us think of the world in Newtonian terms: a global clock with objects continuously changing their states governed by, say, differential equations that express the advancement of time. However, this intuitive and clear time-driven concept often generates inefficient computer algorithms. Specifically, each component of the system under study typically changes its state rarely and such changes are asynchronous among the components. If we make a snapshot of the model at a random time instance, we will see no or few changes that are taking place. The changes are sparsely "sprinkled" over the space-time. If the computer simulates such a system in a time-driven fashion by continuously monitoring each component, the processing power of the machine is wasted. Using a less intuitive but more efficient discrete event model we can simulate the same system orders of magnitude faster on the same computer.

In a discrete event model, the system and its components change their states instantaneously at discrete times; those changes are called *events*. The state remains constant on the intervals between the events. Time advancement is represented not in a time-driven form, like differential or difference equations, but in an event-driven form. In the latter form, the trajectory of the system is a directed acyclic *event dependency graph*. The nodes of it are the events, and the links represent cause-effect relations in pairs of events.

This graph can be also viewed as a data-flow diagram. To each node-event corresponds the event *descriptor*, which consists of the time of the event and the specification of the state change represented by the event. If events $e_1, e_2, \ldots, e_k$ are all the immediate causes of event e, then the descriptor of event e is a function of descriptors of $e_1, e_2, \ldots, e_k$. Computing the descriptor of e given the descriptors of e's causes is called *processing* of event e.

Not only the event descriptors but also the topology of the event dependency graph is unknown in advance. The discrete event simulation algorithm must both construct the event dependency graph and process the events on it. The former activity, also referred to as event *scheduling*, typically is the most calculation intensive and difficult for programming.

An example is helpful. Consider a gutter, bounded from both ends. The gutter contains $N = 4$ balls of equal mass and size. Assume the balls and the gutter walls are ideally rigid and elastic, the gutter is very massive, and its width is just enough to assure motions of the balls along its length in the absence of gravitation and friction. Figure 1 represents the initial segment of the system trajectory beginning with the positions and velocities of the balls at time t_0.

The depiction can be viewed both as a space-time diagram of the system trajectory and as an event dependency graph where the events e_i, $i = 1, 2 \ldots$ are ball-ball or ball-wall collisions. (The gutter and the four balls are depicted at the bottom of the diagram.) An event descriptor here consists of: the collision time, the positions and velocities of the involved balls immediately after the collision. One can check, for instance, that (the descriptor of) event e_5, which is a collision of balls 3 and 4, is a function of (the descriptors of) events e_4, a collision of 2 and 3, and e_2, a reflection of 4 from the wall. Specifically, we can compute time t_5 of collision e_5, if we know the times, positions, and velocities of balls 3 and 4 immediately after their previous events e_4 and e_2, respectively.

It was said above that the state of a discrete event system remains *constant* on the intervals between events. However, in this example positions of the balls are continuously *changing* between collisions. Is there a contradiction? No, there isn't. We simulate the given continuous time system *as if* the state remained constant between collisions. The "trick" is that although we do not consider trajectories of balls between collisions, no information is lost. This "trick" is exactly the reason why simulating the billiards in the discrete event format is much more efficient than in the continuous time format. In the latter, time-driven method, the computer updates the state of the system by scanning all balls at each small interval Δt. In most such intervals, like in the one indicated in Figure 1, there is simply no events, in others there are few events. By contrast, in an event-driven method, the computer updates the state of the system from collision to collision, i.e., from t_i to t_{i+1}. For example, immediately after processing collision e_4 of balls 2 and 3 the computer processes collision e_5 of 3 and 4.

To process event e_5 we must first make sure that e_5 takes place. This is the task of *scheduling*. This task is not trivial even in our simple example. Here is a "naive" way to schedule event e_5: First, we calculate the state of all balls at time t_4 of the last processed event; then from this state we try to match any

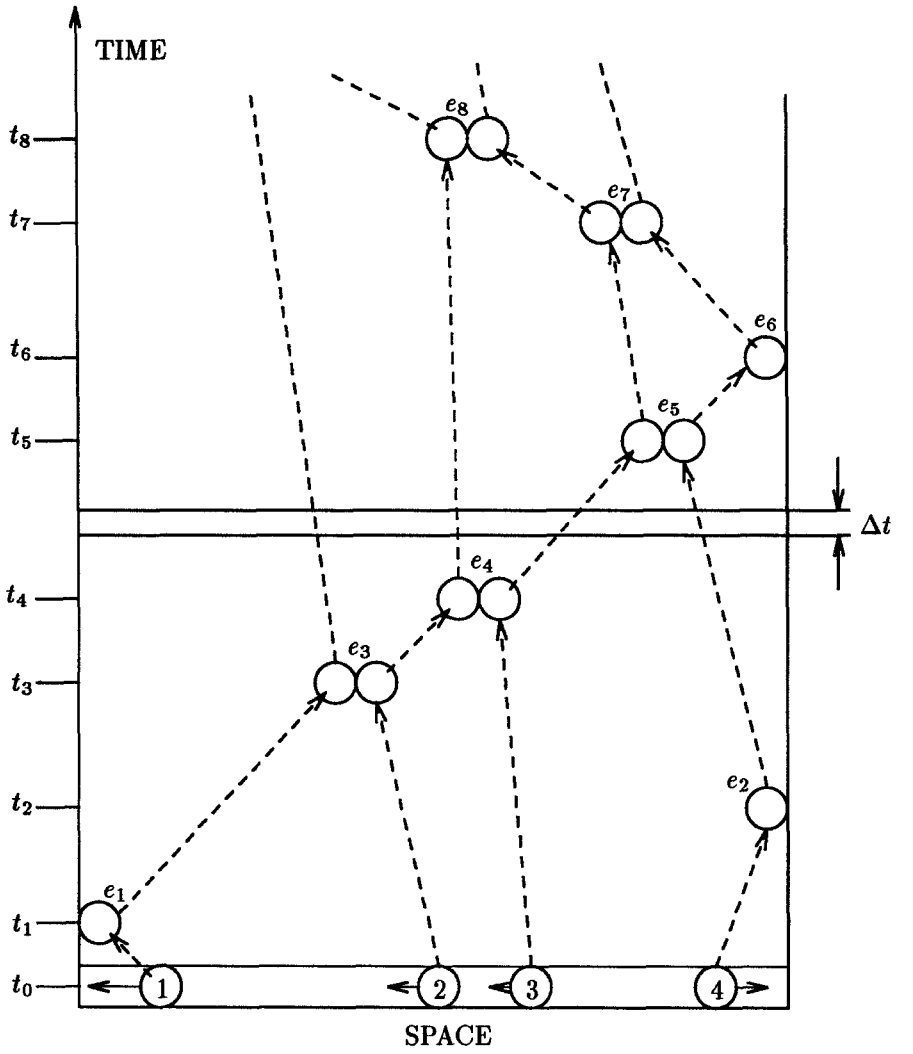

Figure 1: Billiards in one dimension

ball-ball or ball-wall pair in a next collision; lastly, we find the minimum of all these next potential collision times which delivers for us the time of event e_5 and also identifies the participating balls, 3 and 4. After the schedule for event e_5 is completed, we process the event by recomputing positions of balls 3 and 4 and by determining their new velocities after collision e_5. We use known formulas of elastic collision in recomputation of velocities.

The described method of scheduling is "naive" not only because it does not take advantage of 1D (balls can not "overtake" each other, hence only balls i and $i+1$ can collide and only balls 1 and N can collide with the walls), but also because it examines all N balls in order to schedule only one event which involves at most two balls.

The task of designing an efficient scheduling usually constitutes the main difficulty of recasting a continuous time model in a discrete event format. Discussions of efficiency of discrete event simulations, are usually reduced to efficiency of implementing *queues of events*. One should realize, though, that handling such a queue, even if it is done efficiently, does not cover the entire task. Simulationist also has to design an efficient data manipulation, to figure out what *are* the events, which events to keep and which to forget and when during computations.

Say, in the billiards example, which data should be kept for each ball at each stage of computing? Should the entire prehistory of a ball be retained or only a part of it? How to avoid the order of N overhead of the "naive" method? There seems to be no general recipe for answering questions of this type and we are not going to discuss them in this tutorial. (In the specific example of simulating billiards, several methods equivalent in its results to the "naive" method, but which are much more efficient, are mentioned in Bibliography.) It will be assumed here that the discrete event model of a system subject to simulation is defined and is provided with the required mechanism for scheduling future events.

2 Parallel discrete event simulation: its intent and its caveats

The parallel discrete event simulation has the same objective as the serial one. It is supposed to generate the simulated system trajectory in the form of the event dependency graph. An obvious, but sometimes forgotten requirement is that the trajectory, including the topology of the graph and the descriptors of events in it, does not depend on the method of computation, whether serial or parallel. As the programmers know, writing a parallel code is always more difficult than writing a serial code for the same task. Then why do we get involved in parallelization? For one reason only: a promise of shorter running time.

To reduce the running time by parallel execution, a simulation task has to be split into a number of subtasks which can be carried concurrently by different processing elements (PEs) of the parallel computer. Three methods of splitting are commonly used with different PEs carrying:

1) independent simulation runs. This is called the *replication* method

2) different tasks in the serial processing of one simulation run. This is the method of *functional parallelism*

3) different components of the simulated system for one simulation run. This is the *space-parallel* method.

Recently a fourth method has been discussed, where different PEs concurrently carry simulation for different time segments of the same simulated component for one simulation run. It is natural to call it the *time-parallel* method.

The replication method is simple and efficient and should be exercised whenever possible, specifically, when we have to run many independent trajectories with initial conditions known in advance and when one PE is able to accommodate one run. Unfortunately, these conditions are satisfied rarely. Usually we need to speed up a single run or each run in a *sequence*.

The functional parallelism allows one to speed up a single run. In our billiards example in Section 1, the functions that can be executed concurrently may be: solving an equation to find the time of next collision of two given balls, computing the ball velocities after the collision, various data manipulations involved in scheduling the next event, such as the minimization. In addition to being to some extent independent in handling one event, these functions are somewhat independent when executed for successive events. Hence their execution can be pipelined, i.e., we may begin scheduling next collision while still processing the previous one.

The drawback in a functional parallelization is that the degree of parallelism does not scale with the size of the simulated system, e.g., with the number of balls. In scheduling the next event, there is always a serial section of the code. Even if simulation is pipelined, only a limited number of events are taken for processing at a time. Hence, functional parallelism as such is not appropriate for massively parallel execution (but can perhaps be used in combinations with other methods discussed below).

Only the third and the forth methods seem appropriate for massively parallel execution and we will discuss them in detail. Consider the third method, when different components or subsystems are hosted by concurrently running PEs. Recall that our concern is to simulate event dependencies correctly.

Let us first consider the space-parallel method in the application to the (notably inefficient) time-driven simulation. In this setting, at each step the PEs, based on the information about the events processed for times before t, process events for slot $[t, t+\Delta t)$. Because Δt is assumed to be very small it is not probable for both cause and effect events to fall into the same slot. Hence violations of causality almost never occur: between the steps the PEs will inform each other about the causes to correctly schedule the effects.

One source of inefficiency here is *statically fixed* Δt which thereby must be very small. We can improve efficiency by letting Δt *change dynamically* from iteration to iteration. We can choose Δt as large as possible at each new iteration with the restriction that no violation of causality occurs. Specifically, causality

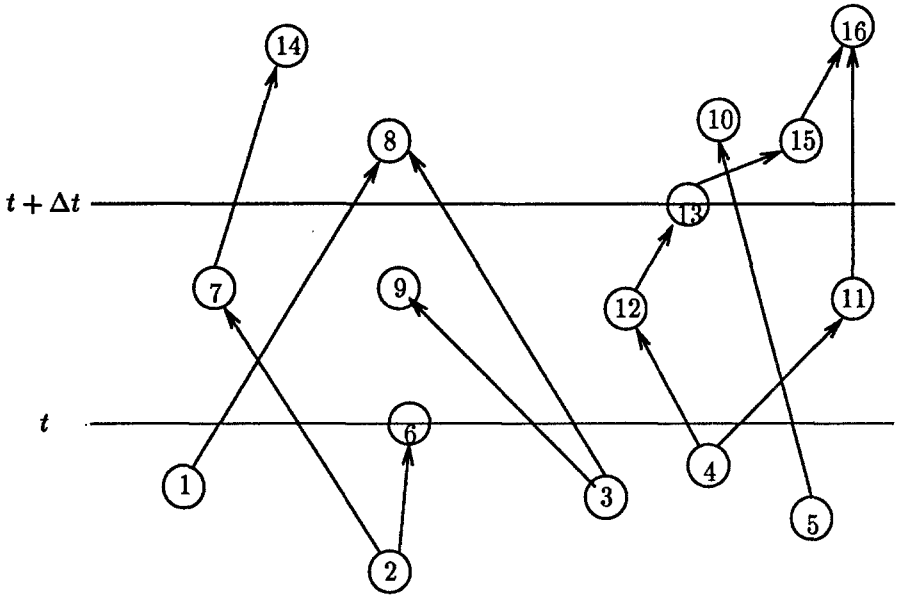

Figure 2: Adjustable time stepping algorithm

is preserved, if given that the smallest event time scheduled but not processed at previous iterations is t, any other such event (which is scheduled but not processed at previous iterations) whose time is smaller than $t + \Delta t$ is itself not an effect of such an event. We choose Δt to be the largest value that satisfies this condition.

This idea is illustrated in the fragment of the event dependency graph in Figure 2. It is assumed in this depiction that all dependency links among the shown events are also shown, and that no shown event will be canceled or rescheduled. (These assumptions are made for all the other pictures in this tutorial unless stated otherwise.) Here the events that are indexed 1, ...5 have been processed, the rest, events 6, ...16, have been scheduled but not processed. Of the latter set, event 6 has the smallest time t, all events in set $S = \{6, 7, 8, 9, 10, 11, 12\}$ are scheduled and unprocessed, and in addition are not effects of events in S. The key observation is that the events in S can be processed concurrently without violating causality. Event 13 has the smallest time among those that are *not* safe to process in parallel with events in S; indeed, 13 is an effect of event $12 \in S$. Events 13,14,15,16 will be processed at future iterations. Moreover, because of the time stepping restriction, events 8 and 10 will be processed at future iterations too, despite that it is safe to process them now. The next Δt is the maximum width of the strip that does not contain events which do not belong to S. The next iteration will begin with $t + \Delta t$ replacing t.

This idea was successfully implemented in several massively parallel simulations. Its use is hinged on finding a convenient method to generate non-trivial (i.e., not very small) estimates for cause-effect delays at least two cause-effect

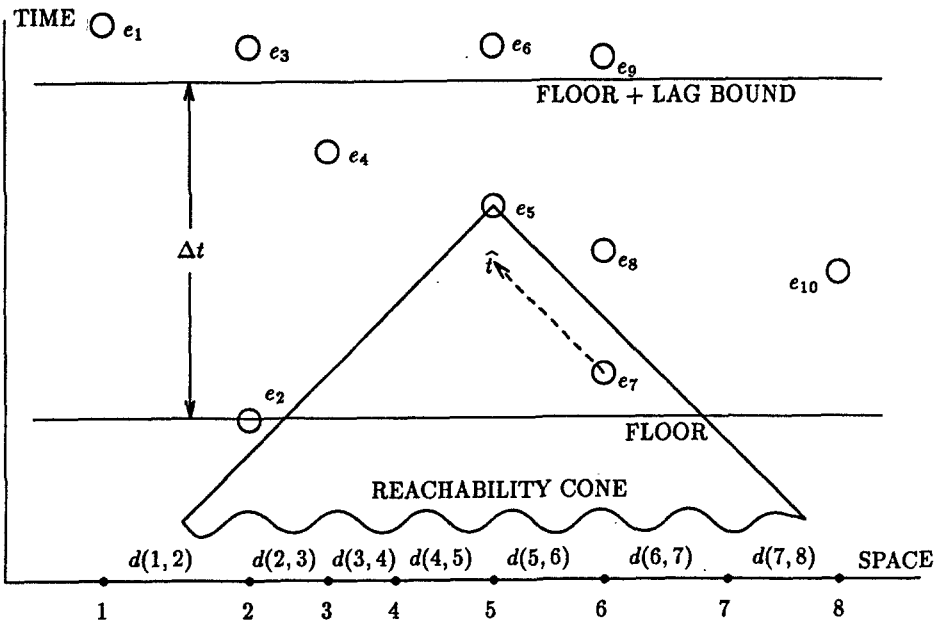

Figure 3: Bounded lag algorithm

links in advance. A success in this, of course, depends on an application (see Bibliography).

Another application-dependent method to preserve causality is based on estimates of *minimum propagation delays*. Here we also advance time by time stepping. However, the width Δt of the strip in the space-time diagram is not thought of as "small." On the other hand, an iteration does not necessarily exhaust all the events that fall within the strip. This is the *bounded-lag* algorithm. An instance of event scheduling and processing is shown in the space-time diagram in Figure 3. Eight sites indexed 1, ...8, are depicted along the space axis. It is assumed that events can not propagate from site i to site $i+1$ in time smaller than $d(i, i+1)$. Quantity $d(i, j)$ is induced for any pair of sites i and j so as to satisfy the triangle inequality $d(i, j) + d(j, k) \geq d(i, k)$. Quantity $d(i, j)$ is also non-negative, but unlike the standard definition of the *distance*, $d(i, j)$ is not necessarily equal to $d(j, i)$. To simplify the drawing, equality $d(i, j) = d(j, i)$ is also assumed to hold in the example of Figure 3, and it is also assumed that $d(i, j)$ is the Euclidean distance between the corresponding sites.

Suppose we want to test, whether or not event e_5 is safe to process at the

current step. In the simplest version of the algorithm, we check all event-causes, whose effects might potentially affect site 5 in the past of event e_5. Space-time coordinates of those events have to belong to the *incoming reachability cone* constructed with respect to event e_5. The condition that an event with time t_1 that occurs at site s_1 belongs to the incoming reachability cone of an event with time t_0 that occurs at site s_0 can be expressed as: $t_0 - t_1 \geq d(s_1, s_0)$. For $s_0 = 5, s_1 = 6, t_0 = \text{time}(e_5)$, and $t_1 = \text{time}(e_7)$ this condition holds and we see that e_7 may cause an event at site 5 in the past of event e_5. This hypothetical trouble event would have time $\widehat{t}$.

The events that are safe to process at the current iteration according to this test are e_2, e_7, and e_{10}. For the next iteration, the floor will be moved to $time(e_8)$.

Event propagation delays are easy to think of as physical delays of propagating signals. That may be the case, but more often *procedural* delays qualify as the event propagation delays. For example, the service time in a queuing system simulation can be translated into an event propagation delay. This might seem counter-intuitive, because both the cause, "service start" and the effect, "service end" occur at the same site (at the server node).

The two discussed above algorithms are examples of safe, causality preserving simulations, also called *conservative*. Without further discussing other safe parallel algorithms, we note that a successful realization in parallel for such an algorithm needs non-trivial (i.e., not equal to zero), a priori estimates of cause-effect delays. The "a priori" means that the estimate must be known before the corresponding events are simulated. One way to see how this is possible is to imagine that a cause and the corresponding effect mark the beginning and the end of a certain "activity." Without simulating this activity we should be able to say that the activity would take longer than a certain positive bound.

Simulations of *stochastic* models by the nature of the assumptions usually made in such models open an avenue for such an estimation. For example, if a job enters the service, the service time is usually assumed to be stochastically independent of the state of the system when service begins. Thus, we can pre-sample the service time several steps in advance, even when the corresponding job is not yet arrived for service. Another example: Ising spin simulations. Here we have an array of atoms and the state of each atom is changed at unpredictable random times which we model as a Poisson point process associated with this atom. The rate of arrivals is fixed for all atoms and arrivals for different atoms are independent. The state change depends on the states of the neighboring atoms, but the *time* of the change is independent of the state.

Paradoxically, whereas the times of state changes are random and hence conceptually unpredictable, in the simulation we can predict them. We sample these times any number of steps in advance using algorithmically generated random sequences. The fact that these sequences are fully deterministic (and reproducible once started with the same seeds) is an advantage! It is not appropriate for our purposes in pursuit of the "real" randomness to use physically generated and hence irreproducible random sequences instead. (Practitioners of simulation

may not recognize this simple but important observation.)

However, presampling is not always possible. In some examples, in order to know a non-trivial estimate of the cause-effect delay we must, at the least, simulate both cause and effect events. This is a fundamental problems with the conservative parallel simulation algorithms, which thus can not always be successfully applied. An example of such impossibility for delay prediction is the simulation of billiards discussed in Section 1. Some may consider the simulation of billiards to be a "toy" example. In fact, this "toy" model is in many respects more difficult to simulate in parallel, than some "serious" models, e.g., queuing networks.

3 Space-time paradigm: everyone was skeptical at first

Figure 4 illustrates the space-time relaxation concept for parallel discrete event simulations. According to this concept, space-time is to be split (arbitrarily or as convenient) into regions and each region is to be assigned to a PE which is responsible for filling this region with events.

The computations are iterative. A synchronous version of such computations can be described as follows. Let $X^{(k)}$ denote a trajectory (event dependency graph) as known at iteration k. This $X^{(k)}$ is composed of segments of trajectories known to each PE. For the next iteration $k+1$ each PE updates its segment of trajectory, i.e., reprocesses its events after receiving relevant information from the neighboring PEs about events as they were known to them at the previous iteration. This reprocessing can be expressed as $X^{(k+1)} = F(X^{(k)})$ where function F symbolizes the cause-effect relation among the events.

The recomputation terminates at the iteration at which each PE detects that its events are the same as they were at the previous iteration. The termination is equivalent to finding X such that $F(X) = X$. This X is a *fixed-point* of the cause-effect relation map F.

Reducing problems to solving fixed-point equations is not unusual in mathematics. For example, we may try to solve equation $x = sin(2x)$ with respect to unknown x by iterating: beginning with $x^{(0)}$ we find $x^{(1)} = sin(2x^{(0)})$, then $x^{(2)} = sin(2x^{(1)})$, then $x^{(3)} = sin(2x^{(2)})$, and so on. The standard questions here are: Is the solution unique? If so, will the iterations converge to this solution? If so, how fast? In the $x = sin(2x)$ problem the solution is not unique and the convergence depends on the choice of the initial guess $x^{(0)}$.

On the other hand, in the discrete event simulation problem, the iterations *always converge* and the found fixed-point is *unique*. This can be seen by comparing the iterative fixed-point method to standard serial simulation. In the latter at each step we uniquely determine one more event using event dependency. Beginning with the same initial events as in the serial simulation, at each iteration of the parallel fixed-point method we settle at least one additional event. And since in the parallel fixed-point method we use the same event dependency (represented in F) the settled events must be the same as the ones in the serial simulation.

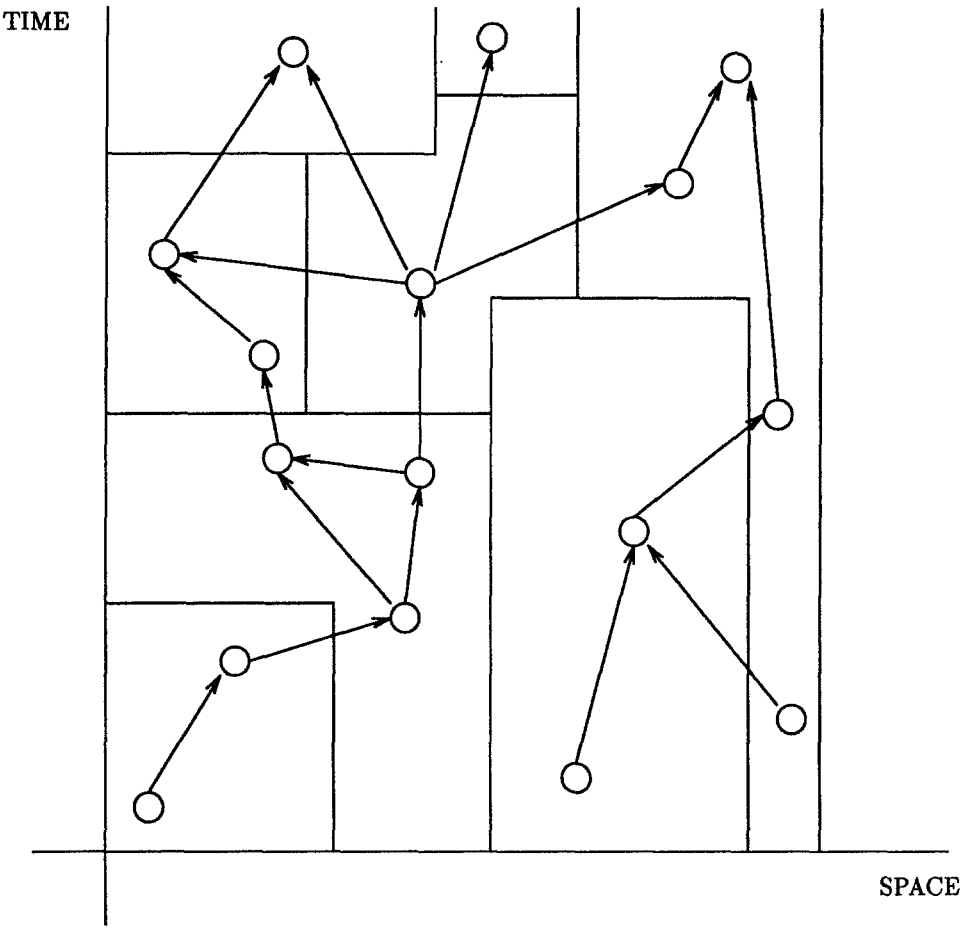

Figure 4: Space-time simulation concept

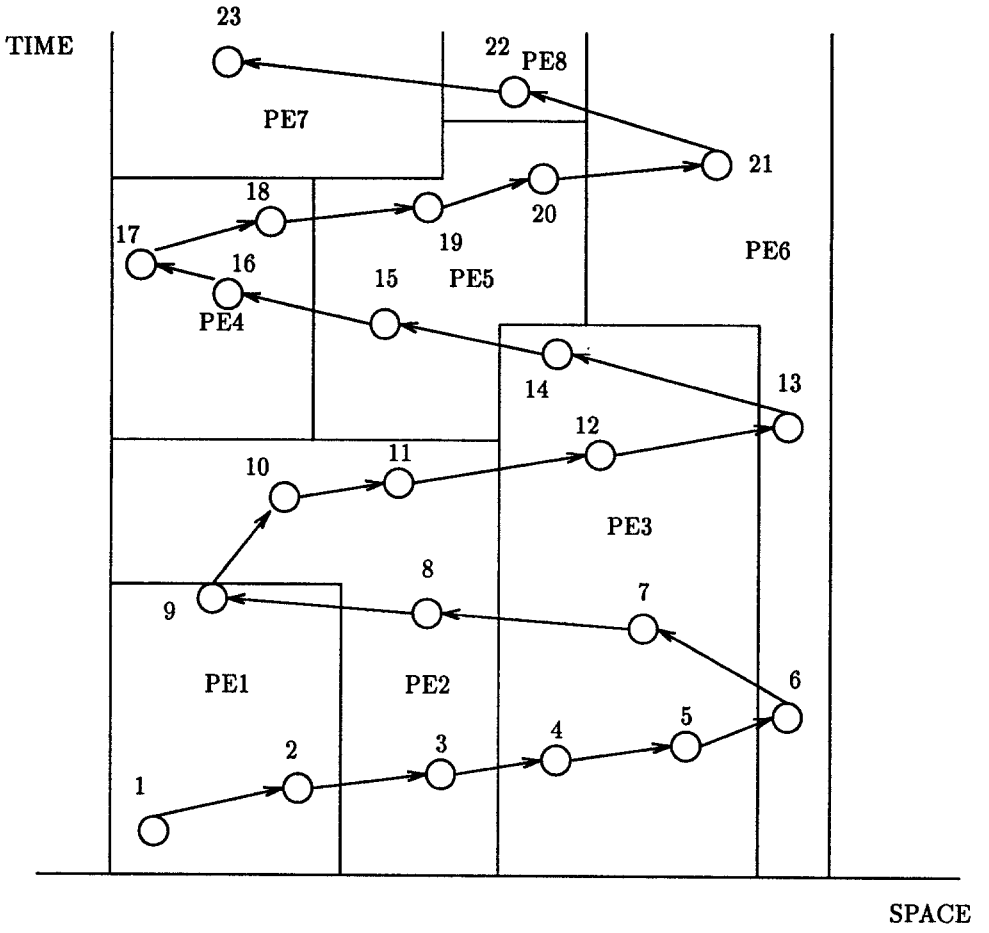

Figure 5: Non-parallelizable event dependency graph

Slow convergence may be an obstacle on the way to a practical realization of this idea. Indeed, for the event dependency graph depicted in Figure 5, no matter how we split the space-time among the PEs, serial event settling (in the shown order: 1,2,3...) is guaranteed: PE6 has to wait for PE3 to correctly determine event 5 before processing event 6 and then wait for PE3 again to correctly determine event 12 before processing event 13 and finally wait for PE5 to correctly process event 20 before processing event 21; similarly, PE 7 can not perform any useful work before events $1, \ldots 27$ are correctly processed by the other processors, and so on.

During the first public presentation of this space-time relaxation paradigm in

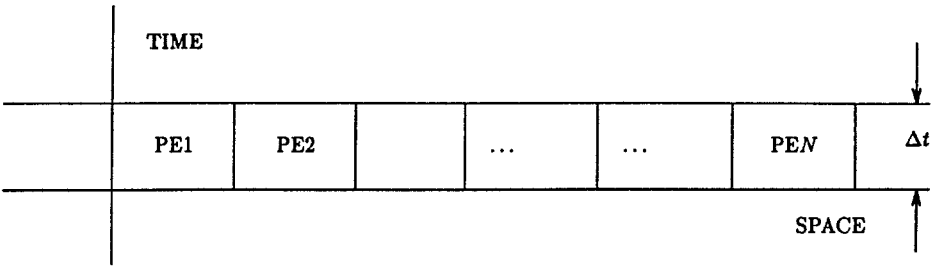

Figure 6: Partitioning of a one-time-step strip

1989 (see Bibliography) the audience seemed very skeptical for this very reason: questionable performance. In 1990 and later the question of performance was addressed as discussed in the following sections.

4 Space-parallel relaxation can be efficient

Here we discuss a specialization of the space-time relaxation idea of Section 3 for a space-parallel simulation. We consider a time-stepping algorithm, where "space" in the space-time diagram represents a large simulated system, like a large queuing network, or a large billiards table with many balls, whereas, the "time" is restricted to a relatively small Δt window. As in the time-stepping algorithms discussed in Section 2, the simulation is advanced by serially processing these strips, one after another. We are now discussing processing events in one specific strip at one step of such an algorithm. Each PE is assigned a subsystem, e.g., a subnetwork, or a region on the billiards table, with the task to process all events on the specified time interval Δt.

This would correspond to a partition of the Δt strip by vertical lines into rectangles as shown in Figure 6. Now if we apply the general iterative procedure described in Section 3 for this specific partition, how many iteration will there be until convergence? This, of course, depends on the event dependency subgraph that fits in the strip. For the event chains like the one in Figure 5 there will be many iterations. However, Figure 5 depicts an artificially difficult, worst case example.

In Figure 7, on the other hand, we do not assume an adversary simulation problem. Depicted here is an "average" example obtained (without thinking of a particular application) by "randomly" sprinkling the events-circles and possible event dependency arrows. How many iterations will be required for the shown event-dependency graph?

It turns out that a good upper bound on the number of iterations can be supplied by counting *levels*. Because the levels can be identified without knowing how the space-time strip is partitioned among the PEs, no partitioning is shown in Figure 7. Level 0 in Figure 7 consists of already processed events that are

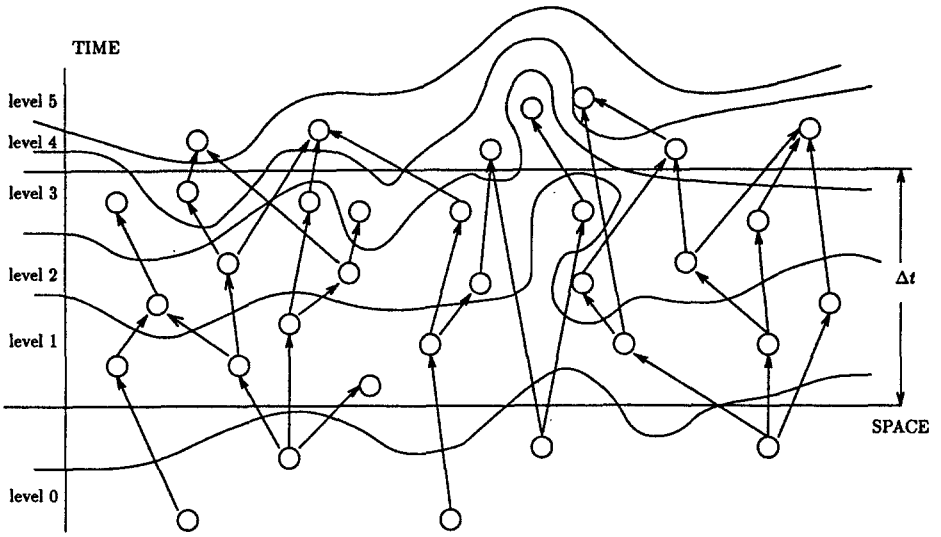

Figure 7: Event dependency levels

positioned below the strip. Level 1 consists of those events at or above the floor of the strip which are direct effects of only level 0 events. By induction, for $k = 1, 2, 3 \ldots$, level k consists of the events at or above the floor of the strip, whose direct causes are level $k-1$ events or lower. For a level k event there must be at least one level $k-1$ event among its direct causes.

At the outset, all level 0 events are correct. After all the PEs process their subsystems once, more events will be correct and all level 1 events at least will be among the correctly settled events. It can be seen by induction that after iteration k of the relaxation procedure all events at level k or lower are determined correctly. Thus, the number of levels (for those events of the event dependency graph that fit within the considered Δt-strip) is the upper bound on the number of iterations needed for correctly determining all events for this strip. One more iteration with the exchange of information may be needed to detect convergence. Actual number of iterations can be smaller than this upper bound for two reasons:

1) initial guesses of events are correct by accident

2) the event dependency subgraph hosted by a processing element contains a

complete set of cause-effects for several levels without need to know events in the neighboring processing elements.

Situation 1 is not always negligibly rare: in the applications in which there are not many choices for an event (e.g., only two choices) reasonable initial guessing might save iterations.

An extreme case of situation 2 is completely independent subsystems hosted by different PEs, or, for that matter, just a single PE which hosts the entire system. In these conditions, all events are determined correctly at the first iteration.

The question remained is: How many event levels fits in the Δt-strip on an "average"? Let N be the size of the simulated system (examples: the number of nodes in the network, the number of billiards balls). We propose a conjecture which says, that, in a "generic" example, if Δt is fixed and N tends to infinity, the "average" number of levels increases not faster than $\log N$.

To investigate this conjecture rigorously one must supply a measure in the space of realizations, thereby assigning an exact meaning to "generic" and "average." Such a measure should express characteristics of the application. This exercise has been performed with some applications (see Bibliography) and, while proving to be not an easy one, confirmed the conjecture.

We will now attempt a superficial but short and easy "proof" of this conjecture irrespective of the application. An *event dependency chain* is a directed path $e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_k$ on the event dependency graph. It can be easily seen that the number of levels in a subgraph of the event dependency graph is the length of the longest event dependency chain in this subgraph. (The length of an event dependency chain is the number of events on it.)

Let us assume that

(a) as N increases the number of event dependency chains increases not faster than proportionally to N

(b) the length of each chain is random and is bounded by distribution from the above with a fixed exponentially distributed random variable

With these assumptions, it can be proven rigorously, that even as different chains are interdependent, the mean value of the maximum of the lengths grows not faster than $\log N$ (see Bibliography).

Assumptions (a) and (b) together bound from the above the amount of the simulated event activity and its spatial non-uniformity. Singular very nonuniform activities, like a fast propagation of a signal through the entire simulated system, e.g., like in Figure 5, are allowed but they must be exponentially rare, as specified in (b).

5 Even time-parallel relaxation may be efficient, when augmented by certain other techniques

Consider the space-time diagram in Figure 8 which is "orthogonal" to that in Figure 6. Here the space interval is thought of as "small," e.g., the simulated

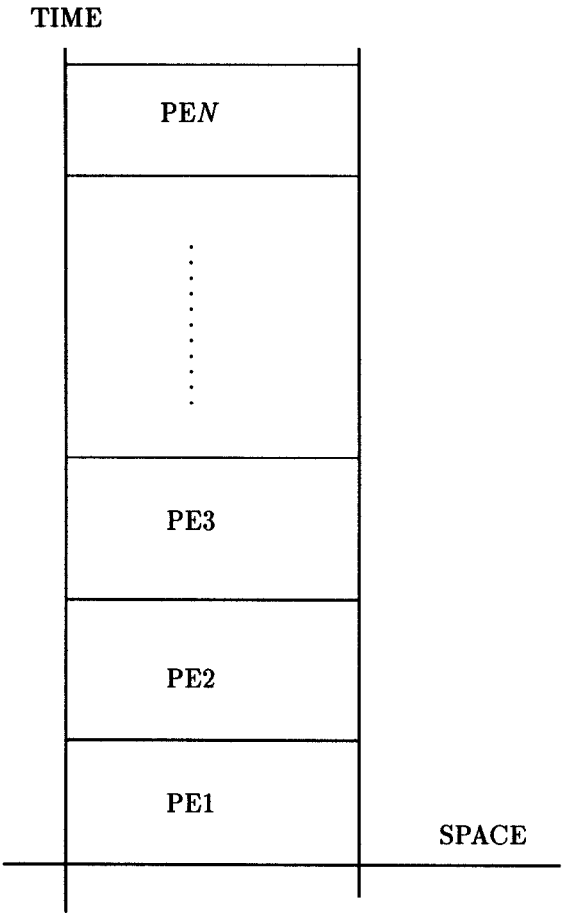

Figure 8: Time-parallel partitioning in a relaxation

system is of a fixed size, but the time interval is "large," e.g., unbounded. One would not expect quick convergence in this case for the same reason as in the case in Figure 5: PE2 is not expected to do useful work until PE1 sends to it the correct information about its events; this needs one iteration; PE3 is not expected to do useful work until PE2 sends to it correct information about its events; this needs at least one more iteration; and so on up to PEN which could only determine its event correctly after iteration N.

One expects the relaxation to converge not faster than in N iterations unless the event dependency graph can be decoupled, as it extends over time, into several independent "regenerative" components. With such an expectation, the following example comes as a surprise.

We simulate a single FIFO queue with feedback as depicted in Figure 9. We assume that each job makes two service demands. Specifically, job i arrives, say, at time A_i, joining the end of the queue, eventually receives its first service, which

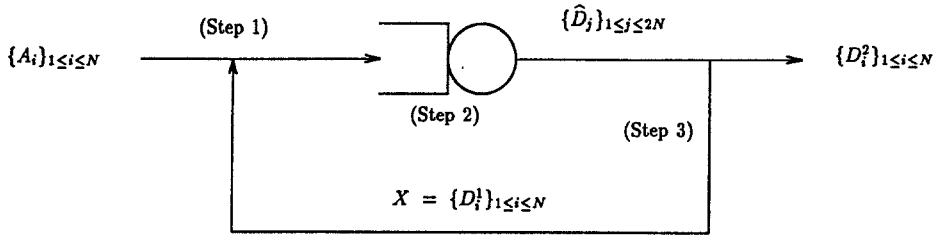

Figure 9: A FIFO queue with a feedback

terminates, say, at time D_i^1, then immediately at the same time D_i^1 reenters the end of the queue, eventually receives its second service, and then finally departs at time D_i^2. (If a job feeds back at the same time that new jobs arrive then, by convention, the new jobs enter the queue ahead of the job feeding back.) Three simulated durations correspond to each job i: time between consecutive arrivals $a_i = A_i - A_{i-1}$ (assuming $A_0 = 0$), first service S_i^1, and second service S_i^2, where $i = 1, 2, ...N$. It is assumed that there are also three corresponding distributions and that each duration is an independent (from system state or other durations) random sample drawn from the corresponding distribution. Thus, the system subject to simulation is a G/G/1 queue with a feedback.

This system fits our assumptions of "small" space interval and "large" time interval as stated above. We assign its simulation to N PEs, so that PEi carries simulation of the time interval that covers the arrival of job i. Exact boundaries between the intervals are not essential; also note that, in the beginning, the assignment of simulated time to PEs is known only implicitly, conditioned to finding the correct events. This distinguishes the presented example from the general scheme in Section 3.

First, the computer samples all random durations. Specifically, PEi obtains a_i, S_i^1, and S_i^2 using its individually seeded random number generator. Second, N values $\{A_i\}_{1\le i\le N}$ are computed, so that PEi obtains $A_i = a_1 + a_2 + ... + a_i$. It takes only one application of the fast *parallel scan* (also called *parallel prefix*) operation. The scan is "felt" like a single programming step. Internally it takes $\log N$ steps of recursive doubling and pointer jumping (see Bibliography).

The final, most involved phase is computing the sequences of departures on the first and second visit, $\{D_i^1\}_{1\le i\le N}$ and $\{D_i^2\}_{1\le i\le N}$. This is done by an iterative relaxation, as discussed in Section 3. Specifically, let X denote the sequence $\{D_i^1\}_{1\le i\le N}$ of first visit departures for the considered N jobs. Function F here can be expressed as a transformation of this sequence into a similar sequence $Y = \{\tilde{D}_i^1\}_{1\le i\le N}$, thus $Y = F(X)$. F consists of three steps along the circular path in Figure 9, namely

Step 1. Merging sequence X with sequence $\{A_i\}_{1\le i\le N}$ of original arrivals. Let Z be the obtained merged sequence of length $2N$.

Step 2. Computing sequence $\{\hat{D}_j\}_{1\leq j\leq 2N}$ of $2N$ departures, given the sequence of arrivals Z.

Step 3. Splitting sequence $\{\hat{D}_j\}_{1\leq j\leq 2N}$ into two subsequences of length N each: subsequence $Y = \{\tilde{D}_i^1\}_{1\leq i\leq N}$ of departures corresponding to external arrivals, and subsequence $\{\tilde{D}_i^2\}_{1\leq i\leq N}$ of departures corresponding to the feedbacks in X.

If we find this X, we can compute the sequence of second visit departures $\{D_i^2\}_{1\leq i\leq N}$ by applying steps 1 and 2 of the procedure used to define F above and yielding in step 3 sequence $\{D_i^2\}_{1\leq i\leq N}$, instead of Y.

Each iteration $X^{k+1} = F(X^k)$ of the relaxation is fast. It employs fast parallel merge at step 1 and fast scan (parallel prefix) for computing at step 2 the departures of the FIFO queues given the arrivals as discussed in the literature (see Bibliography). Step 3 is obviously fast too. It turns out that the number of iterations needed for convergence is also small.

Figure 10 represents an experiment where we fix the termination time $T = 20,000$ and the arrival rate $\lambda = 0.5$ so that the number of original arrivals N is about $T\lambda = 10,000$. The service rate for first and second visit is taken the same, and we vary this common rate μ. Several typical interarrival time and service duration distributions are tried such as singular (constant value), uniform, discrete, and exponential. Figure 10 shows the convergence for exponential distributions (the results for other distributions are similar). Here for each μ we simulate 10 differently seeded random samples; the average value of the number of iterations as a function of μ is represented by a solid line, the upper and lower 99.99% Student's confidence bounds are shown by vertical bars. The convergence is the worst around $\mu = 2\lambda$. Yet it takes less than 30 iterations for $N = 10,000$ arrivals (20,000 events).

This fast convergence is not only experimentally observed but can also be theoretically explained (see Bibliography). The most counter-intuitive case is when $2\lambda > \mu$, that is, of an unstable system. In this case a permanent queue is formed which eliminates the possibility of forming several regenerative subgraphs in the event dependency graph. This method works for a general queuing network also (see Bibliography).

6 Time Warp simulation: where it fits in our scheme of things

The Time Warp algorithm for parallel discrete event simulations has been widely popularized since 1985 (See Bibliography). The TW is a rollback-based algorithm (such algorithms are also called *optimistic*), that is, it allows each PE to process as many scheduled events as it can even without full assurance that these events are correct, thus avoiding the difficulties of event scheduling as discussed in Sections 1 and 2. Incorrectly processed events are corrected later by rolling the simulation time back and reprocessing.

The novelty of TW is its way of making these rollbacks: each PE maintains a queue of events (or "messages" in the original TW formulation) and tries

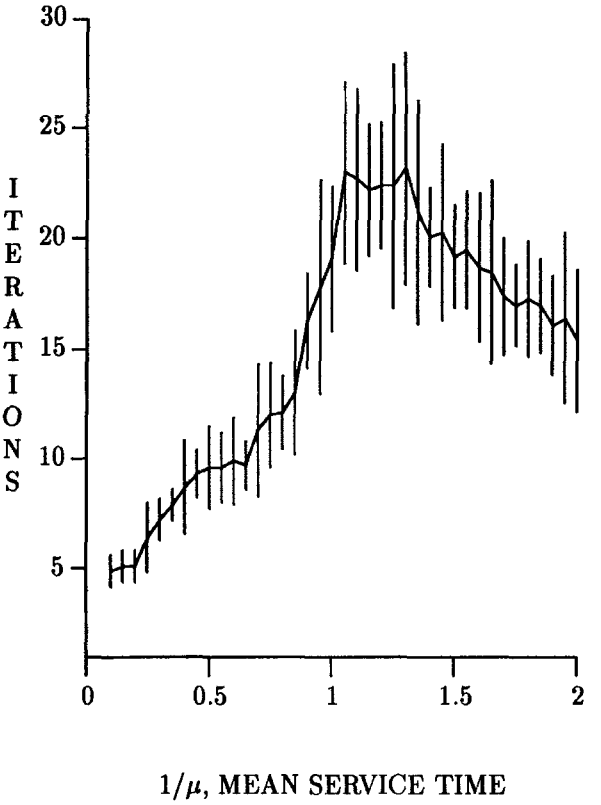

Figure 10: Convergence for the simulation of a queue with a feedback

to process them in the time increasing order thereby scheduling future events for itself and other PEs. Events can be "positive," i.e., normal events, and "negative." Negative events are the instrument of rollback. A negative event $-e$ is generated by a PE in the process of un-doing the corresponding positive event e. This $-e$ is generated when the PE receives an evidence that previously sent out event e was incorrect. Event $-e$ is being sent out in the same way and to the same PEs to which event e was sent. The hope is that $-e$ catches up with e. Specifically, it may happen that this $-e$ finds its counterpart e not yet processed. When such a match of two unprocessed events $-e$ and e is detected, it means that $-e$ catches up with its positive counterpart e. Then both are erased from the event queue ("annihilated"), thereby terminating the "lineage" of wrong events. If $-e$ fails to catch up with e, because the PE has already processed e and sent its effects, then the "lineage" continues. Say, events e_1 and e_2 were generated as the effects of event e and were sent to other PEs. In this case, the PE that did this erroneous processing of e must similarly process $-e$ and must similarly send its effects $-e_1$ and $-e_2$ to the corresponding other PEs in the hope that they in turn would catch up with their positive counterparts.

Programmers see in the TW an ingenuous cancellation strategy. However,

computationally the TW is just an *asynchronous relaxation* as the other rollback-based algorithms. Thus, instead of reprocessing events by each PE, maintaining common iterations, a PE in TW reprocesses its events at its own pace, without explicitly synchronizing with other PEs. In the synchronous relaxation, as described above, all PEs iterate over a specified batch, which may be a set of events or the time interval, until every PE detects convergence for this batch. Then all PEs start processing next batch. In TW, on the other hand, there is a notion of *global virtual time*. The GVT is the virtual (i.e., simulated) time below which no PE can rollback. Generally, a PE processes ahead of the GVT mark, so that the converged events are those with time smaller than GVT.

There are more "degrees of freedom" in TW and other asynchronous relaxation algorithms than in a synchronous relaxation algorithm. For example, there are "aggressive" and "lazy" versions in TW. In the former, the cancellation by sending antievents is done each time a rollback takes place. In the latter, the antievents are only sent for those events which are turned out to be wrong as seen during reprocessing. The hope in a "lazy" cancellation is that despite some intermediate errors, the final results were still correct. Obviously, there is no similar subdivision in the synchronous relaxation and at each iteration all events are reprocessed, at least virtually. (Some optimization in the flavor of "lazy" cancellation is still possible which would reduce inter-PE communication traffic, but it will not change the number of iterations to convergence.)

As a result of its tighter synchronization, synchronous relaxation behaves better in its worst case, than TW in its worst case. Long non-parallelizable event chains are the only known reason for slow convergence of synchronous relaxation. On the other hand, there are examples when TW introduces cascading and slows down unduly even for well parallelizable models.

Load disbalance, when some PEs have many more events to process than the other PEs can slow down each iteration of synchronous relaxation. Superficially, load disbalance seems not to be a problem for TW, as a lightly loaded PE is not explicitly restricted in advancing its local time. However, this would lead to a large discrepancy in local "virtual" times among the PEs which, as practice shows, slows down the computations significantly. Thus, the load disbalance is as much a problem for TW as for synchronous relaxation.

Synchronous relaxations is especially well suited for SIMD processing and since such machines with thousands of PEs are available, the synchronous relaxation algorithms have been implemented in examples. On the other hand, TW needs a MIMD parallel machine and since massively parallel MIMD computers are lagging in their commercialization as compared with SIMD computers, there has been no report yet of an efficient TW implementation for a computer with thousands of PEs. There are some reasons to suspect that in certain cases for thousands of PEs, the TW, if unprotected by additional mechanisms, may fail in performance due to cascading and echoing phenomena (see Bibliography).

7 Conclusion

The discussion of massively parallel discrete event simulation in this tutorial has been centered around the task of designing its computational engine. It is recognized here that by simply adapting the existing computation engine of serial simulation, which is an event list, no substantial progress can be achieved in the task of efficient massively parallel simulation. There is more to a vehicle than just an engine. For example, the *object oriented paradigm* of programming has recently become fashionable among simulationists. One should realize though that these techniques as well as recent advances in the *graphical user interface*, no matter how useful, can not improve performance of processing in simulation, when massive parallelism is concerned. Whether or not the task is performed efficiently is mostly determined by the mathematical properties of the underlying computational technique. Mostly, this tutorial discussed these algorithmic techniques. Among them relaxation appears the most promising one for the task. It is also amenable to implementations on currently available SIMD and SPMD massively parallel computers. In applications for such machines, speed improvement ratios in the hundreds have been obtained (in comparisons to fast work stations), while self-speedups (speed improvements with respect to a single PE of the same computer) have been in thousands.

Bibliography

Section 1

R.E. Shannon, "Introduction to Simulation," in *Proceedings, 1992 Winter Simulation Conference*, 65-73.

Presents a wider view on the simulation activity than the one accepted in this tutorial (constructing a dynamic system trajectory). One of many treatises on the subject.

D.C.Rapaport, "The Event Scheduling Problem in Molecular Dynamic Simulation," *Journal of Computational Physics*, Vol. 34, No.2, 1980.

Describes an algorithm for a serial billiards simulation. Uses a complex list structure to maintain all feasible scheduled collisions for a ball.

B.D. Lubachevsky, "How to Simulate Billiards and Similar Systems," *Journal of Computational Physics*, Vol. 94, No.2, 1991.

Introduces an alternative serial billiards simulation algorithm. Uses a simplified data structure: only two events per ball, one past event and one future event. This appears not less efficient that Rapaport's algorithm.

D.C.Rapaport, "A Note on Algorithms for Billiard-Ball Dynamics," *Journal of Computational Physics*, Vol. 105, No.2, 1993.

The note compares the algorithms by Rapaport and Lubachevsky for simulation of the billiards.

B.D.Lubachevsky, "Which Algorithm is Better?" *Journal of Computational Physics*, Vol. 105, No.2, 1993.

A comment on the note by Rapaport.

Section 2

P. Hontales, B. Beckman, et al., "Performance of the Colliding Pucks simulation on the Time Warp Operating systems" in *Proceedings, 1989 SCS Multiconference on Distributed Simulation, Simulation Series* (Society for Comput. Simulation, San Diego, CA, 1989), Vol. 21, No.2.

Presents serial and parallel billiards simulation based on Time Warp message queues paradigm.

B.D. Lubachevsky, "Simulating Billiards: Serially and in Parallel," *International Journal in Computer Simulation*, Vol. 2, 1992

Copes with unavoidable errors in parallel billiards simulations using a method different from Time Warp.

D. Nicol, "Conservative Parallel Simulation of Priority Class Queuing Networks," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 3, No. 3, May 1992, 294-303.

Introduces the adjustable time-stepping algorithm as applied to parallel queuing networks simulation.

B. Gaujal, A. Greenberg, D. Nicol, "A Sweep Algorithm for Massively Parallel Simulation of Circuit-Switched Networks," *Journal of Parallel and Distributed Computing*, August 1993 (to appear).

Applies the adjustable time stepping paradigm to simulating long-distance telephone networks.

D. Nicol, A. Greenberg, B. Lubachevsky, "MIMD Parallel Simulation of Circuit-Switched Communication Networks," in *Proceedings, 1992 Winter Simulation Conference*, p. 629-636.

Reports implementations of the long-distance telephone network simulations on the Intel Touchstone Delta MIMD parallel computer. Utilizing up to 256 PEs the processing speed of up to 8 million telephone calls per minute is achieved.

B.D. Lubachevsky, "Bounded Lag Distributed Discrete Event Simulation," in *Proceedings, 1988 SCS Multiconference on Distributed Simulation, Simulation Series* (Society for Comput. Simulation, San Diego, CA, 1988), Vol. 19, No.3.

Introduces the bounded lag algorithm.

B.D. Lubachevsky, "Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks," *Communications of the ACMΦ*, Vol. 32, No.1, 1989.

A more extended discussion of the bounded lag algorithm.

Section 3

M. Chandy and R. Sherman, "Space, Time, and Simulation," in *Proceedings, 1989 SCS Multiconference on Distributed Simulations, Simulation Series* (Society for Comput. Simulation, San Diego, CA, 1989), Vol. 21, No.2.

The original presentation of the space-time relaxation paradigm in parallel discrete event simulation.

Section 4

A.G. Eick, A.G. Greenberg, B.D. Lubachevsky, and A. Weiss, "Synchronous Relaxation For Parallel Simulations With Applications to Circuit-Switched Networks," in *Proceedings, 1991 SCS Multiconference on Distributed Simulations, Simulation Series* (Society for Comput. Simulation, San Diego, CA, 1991), Vol. 23, No.1.

The original presentation of the space-parallel synchronous relaxation. For the specified example (long-distance circuit-switched telephone networks), performance is analyzed mathematically, in particular, it is proven that the number of iterations to convergence for simulating a Δt-strip grows as $\log N$ where N is the number of links in the networks.

T.L. Lai and H. Robbins, "Maximally Dependent Random Variables" *Proc. Nat. Acad. Sci. USA*, Vol. 73, No. 2, 286-288 (Statistics).

Proves, in particular, that the mean value of the maximum of N dependent random variables grows not faster than $\log N$. The result is cited in the tutorial to explain a $\log N$ convergence of the space-parallel relaxation.

Section 5

A.G. Greenberg, B.D. Lubachevsky, and I. Mitrani, "Unboundedly Parallel Simulations Via Recurrence Relations," in *Proceedings, Conference on Measurement and Modelling of Computer Systems* (SIGMETRICS, Boulder, CO, 1990), Vol.18, No.1.

The first demonstration that time-parallel simulation can be efficient. Introduces fast algorithms for solving recurrence relations and relaxation for discrete event simulation on massively parallel processors. The latter algorithms are discussed in the following three papers

R.E. Ladner, and M.J Fisher, "Parallel Prefix Computation," *Journal of the ACM*, Vol. 27, 1980, pp 831-838.

C.P. Kruskal, "Searching, Merging, and Sorting in Parallel Computation," *IEEE Trans. Comput.*, TC-32 (1983), 942-946

Discusses, among others, a fast parallel merging which is used in the algorithm for simulating a queue with a feedback.

A.G. Greenberg and B.D. Lubachevsky, "A Simple Efficient Asynchronous Parallel Prefix Algorithm," in *Proceedings, 1987 International Conference on Parallel Processing* (Penn State Univ., 1987), pp. 66-69.

A.G. Greenberg, B.D. Lubachevsky, and I. Mitrani, "Algorithms for Unboundedly Parallel Simulations," *ACM Trans. on Computer Systems*, Vol. 9, No. 3, 1991

Extended version of the SIGMETRICS' paper of the same authors. Additionally shows that in simulating an unstable queue with a feedback the number of iterations to convergence grows as $\log N$ where N is the number of jobs.

A.G. Greenberg, B.D. Lubachevsky, and I. Mitrani, "Superfast Parallel Discrete Event Simulations," unpublished.

More discussion is provided on the reasons for fast convergence of time-parallel queuing network simulations. More examples of application of the relaxation techniques to parallel discrete event simulation are given (priority FIFO queues, slotted ALOHA protocol).

Section 6

D.R. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, 1985.

The original presentation of the Time Warp algorithm. The system of distributed simulation that include message queues with messages and antimessages is described.

J.D. Biggins, B.D. Lubachevsky, A. Shwartz, and A. Weiss, "A Branching Random Walk with a Barrier," *The Annals of Applied Probability*, Vol. 1, No. 4, 1991.

Studies a mathematical model used in the analysis of the rollback-based algorithms.

B.Lubachevsky, A.Shwartz, and A. Weiss, "Rollback Sometimes Works ... if Filtered" in *Proceedings, 1989 Winter Simulation Conference*, 630-639.

Introduces possible failures modes of rollback algorithms (cascading and echoing) and describes the means to counter these modes.

B.Lubachevsky, A.Shwartz, and A. Weiss, "An Analysis of Rollback-Based Simulation" *ACM Trans. on Modelling and Computer Simulation*, Vol. 1, No. 2, 1991.

An extended version of the previous paper. Contains full mathematical proofs and discussion, including examples on each mode of rollback failure.