# POSE: Getting Over Grainsize in Parallel Discrete Event Simulation

Terry L. Wilmarth and Laxmikant V. Kalé
*Department of Computer Science*
*University of Illinois at Urbana-Champaign*
{*wilmarth, kale*}*@cs.uiuc.edu*

## Abstract

*Parallel discrete event simulations (PDES) encompass a broad range of analytical simulations. Their utility lies in their ability to model a system and provide information about its behavior in a timely manner. Current PDES methods provide limited performance improvements over sequential simulation. Many logical models for applications have fine granularity making them challenging to parallelize. In* POSE*, we examine the overhead required for optimistically synchronizing events. We have designed an object model based on the concept of virtualization and new adaptive optimistic methods to improve the performance of fine-grained PDES applications. These novel approaches exploit the speculative nature of optimistic protocols to improve single-processor parallel over sequential performance and achieve scalability for previously hard-to-parallelize fine-grained simulations.*[1]

## 1. Introduction

Simulation makes it possible to analyze systems that would be expensive, dangerous or impossible to construct prototypes for. Some simulations are too complex for sequential simulation due to space and time limitations. Parallelization partitions large problems to fit in memory on many processors and should reduce execution time, but this has proved to be a challenging problem. Fujimoto suggests that a "sufficiently general solution to the PDES problem may lead to new insights in parallel computation as a whole" [6].

We present POSE, a Parallel Object-oriented Simulation Environment in which we have studied the major obstacles to effective parallelization of discrete event models. In particular, we have focused our study on fine-grained simulations which have exhibited the poorest performance.

[1]This work was supported in part by the National Science Foundation (NGS 0103645) and the local Department of Energy ASCI center, CSAR (B341494).

### 1.1. Parallel discrete event simulation

A discrete event simulation has a *state* that changes at discrete points in time (*timestamps*) via *events.* An *event list* is a queue of events to be performed on the state. A discrete *global clock* keeps track of progress in simulated time. Events are selected from the event list for execution based on the smallest timestamp. When an event is executed, it may schedule future events that are added to the event list. Thus there exist *causality* relationships between the events in the list.

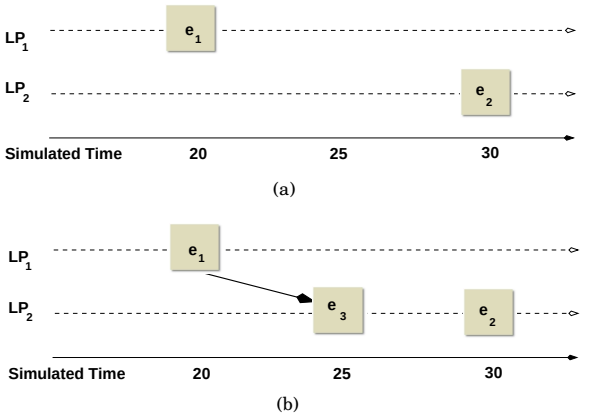


**Figure 1. Causality error**

In parallel, a discrete event model of a system maps physical processes to logical processes (LPs), each with access to a local portion of the state. A local clock keeps track of the progress of the LP. Errors arise when a causality relationship crosses the boundary of an LP.

For example, consider $LP_1$ and $LP_2$ with events $e_1$ and $e_2$ as in Figure 1. Let $T(e)$ be the timestamp of an

event $e$. It would seem to be safe to execute the two events simultaneously. Suppose, however, that $e_1$ issues an event $e_3$ for $LP_2$, and $T(e_3) < T(e_2)$. Then $e_3$ must execute before $e_2$ because it may modify the local state which is later accessed by $e_2$.

### 1.2. Synchronization methods

An LP, on its own, cannot know if the earliest available event is *the* earliest event it will ever have. Methods for synchronizing the execution of events across LPs are necessary for ensuring the correctness of the simulation. There are two categories of such mechanisms.

The *conservative* approach avoids causality errors by determining the safety of processing the earliest event on an LP. An event is safe to process when no other event can generate earlier events on the same LP. Determining safety could lead to deadlocks, so deadlock avoidance or detection and recovery must be employed. Because an unsafe earliest event causes an LP to block, conservative methods are limited in the degree to which they can utilize the available parallelism.

The *optimistic* approach allows LPs to process the earliest available event with no regard to safety. Causality errors are detected and handled. When a *straggler* event arrives with a timestamp less than the LP's local clock, the LP is *rolled back* to the point where the straggler should have been executed. To recover state, we *checkpoint* it during *forward execution*. Events spawned by the events we rolled back are sent *cancellation messages* which remove events from an LP's event list. If a canceled event has already been executed, further rollbacks are required. This can cause a *cascade* of rollbacks throughout the simulation.

Optimistic simulations periodically estimate a global virtual time (GVT), the smallest timestamp of all events in the simulation at some point in time. Nothing earlier than the GVT can be rolled back or canceled, so it is safe to dispose of any checkpoints with an earlier timestamp. The process of reclaiming this memory and *committing* the event to history is called *fossil collection*.

We feel that optimistic mechanisms have greater potential for utilizing available parallelism than conservative mechanisms. Rather than blocking to wait for an event to be safe, optimistic approaches perform *speculative computation* that may not need to be rolled back.

### 1.3. Related Work

Much progress has been made in PDES over the last two decades but major problems remain[5]. Simulations developed by parallel computing experts with a deep understanding of the underlying parallel environment (usually those who developed it) perform well while those developed by non-experts perform poorly. The problem of fine-grained simulations (where synchronization overhead overwhelms computation) remains unsolved.

The most well-studied optimistic mechanism is Time Warp, as used in the Time Warp Operating System[7]. Time Warp was notable as it was designed to use process rollback as the primary means of synchronization. Georgia Tech Time Warp (GTW)[4] was developed for small granularity simulations such as wireless networks and ATM networks and designed to be executed on a cache-coherent, shared memory multiprocessor. GTW features a *simulated time barrier* which imposes a limit on how far into the future LPs can advance. POSE has a similar *speculative window*. GTW achieved speedups of 38 using 42 processors for simulating a PCS network.

The Synchronous Parallel Environment for Emulation and Discrete Event Simulation (SPEEDES)[15] was developed with an optimistic approach called *breathing time buckets*. SPEEDES has evolved to include various strategies for comparison purposes and has been used for a variety of applications including military simulations.

Many PDES systems are combinations or hybrids of existing mechanisms. Yaddes[13] allowed a program to be run sequentially, or in parallel using one of three mechanisms with no changes. Maisie[3] requires minor modifications to render a program executable as a sequential simulation, or parallel using conservative or optimistic synchronization mechanisms. Maisie's successor, Parsec[1], also allows for a mix of synchronization protocols amongst the LPs of a program. The best speedup reported for Parsec was nearly 8 on 16 processors for simulating a 3000-node wireless network using the conservative null message protocol.

### 1.4. Performance goals

Let $T_P$ denote single-processor parallel time. Let $T_S$ be the time obtained by our sequential simulator. $T_I$ or *ideal sequential time* is a lower bound on the sequential time. This is the time it would take to execute the simulation if we knew the events and their order in advance. This measure is useful when the program requires more memory than is available on a single processor. To obtain $T_I$, we run the program in parallel, compute the average grainsize and multiply by the number of events executed. As we shall see in Section 3, this is a lower bound on $T_S$ because of an interesting effect of our approach on event grainsize.

Our goal is to improve the scalability of problems with fine granularity. In particular, we seek to reduce *per event overhead* by making efforts to handle single events apply to *groups* of events. We have three subgoals for POSE. First, to achieve a lower *break-even point*, which is the number of processors required to better $T_S$. This improves the applicability of PDES in small parallel computing environments. Second, we wish to achieve near-linear speedups relative to $T_P$. Finally, we should obtain greater *maximum speedups*.

## 2. Fine Granularity and Overhead in PDES

For every event, there is overhead to handle the event in parallel. If the work performed by the event is small, the overhead will far outweigh it. Bagrodia[2] specifies three primary sources of overhead in PDES: partitioning-related, target architecture, and synchronization protocol overheads. We focus on the overhead of optimistic synchronization and have categorized the types of overhead encountered. Figure 5 in Section 4 charts overhead in a POSE program and illustrates the difficulties to overcome.

**Forward Re-execution Overhead** is forward execution time spent *re-executing* events that were previously rolled back.

**Checkpointing Overhead** is the time it takes to checkpoint an object's state. The simplest approach is *full checkpointing* which makes a copy of the state before *every* event. This can be wasteful if the state is large and complex. *Partial checkpointing* checkpoints only the portion of state that may have changed which is difficult to determine in practice. *Periodic checkpointing* only checkpoints before some events. This requires more overhead to reconstruct a state between two checkpoints and makes GVT estimation more challenging.

**Simulation Overhead** is the time it takes to orchestrate the execution, rollback and cancellation of events according to the synchronization strategy. It includes the time spent receiving events, inserting them in the event queue, determining when and what events can be executed, determining if a rollback is necessary or if a cancellation is pending, etc. Simulation overhead is proportional to the number of events in the simulation. Thus, if the number of events is high but the granularity is very fine, the simulation overhead will be large relative to the forward execution time. **Rollback Overhead** occurs when a straggler arrives and we must undo events, send cancellation messages to spawned events and restore a checkpointed state prior to the straggler's timestamp. **Cancellation Overhead** includes the time taken to issue cancellation messages as well as receiving and handling them. POSE prioritizes a cancellation slightly higher than the message to be canceled and performs cancellations before forward execution steps. **Commit Overhead** is fossil collection time and is performed whenever a new GVT estimate is available.

**GVT Overhead** is the time spent gathering, organizing and using data to compute the GVT estimate. GVT computation is especially likely to overwhelm forward execution time in less than ideal simulations. This is probably why there are so many algorithms for GVT calculation in the literature[11, 16, 12]. The algorithm used in POSE operates asynchronously, allowing the simultaneously execution of events. Its execution will dominate in situations where the degree of parallelism is low, but it doesn't force idle time on processors with plenty of work to do, and has resulted in less average overhead per processor than other strategies.

**Communication Overhead** includes message packing/unpacking and prioritized scheduling. This overhead naturally increases with the number of processors, since the percentage of non-local communication rises.

## 3. POSE

We have chosen to implement our PDES environment in CHARM++[8], a C++-based parallel programming system that supports the *virtualization* programming model. Virtualization involves dividing a problem into a large number $N$ of components that will execute on $P$ processors [9]. $N$ is independent of $P$, though ideally $N>>P$. The user's view of the program consists of these components and their interactions; the user need not be concerned with how the components map to processors. The underlying run-time system takes care of this and any remapping that might be done at run-time.

In CHARM++, these components are known as *chares*. Chares are C++ objects with special *entry* methods that may be invoked asynchronously from other chares. Since many chares can be mapped to a single processor, CHARM++ uses *message-driven execution* to determine which chare executes. A dynamic scheduler running on each processor that has a pool of *messages* (entry method invocations), selects one, determines the object it is destined for and invokes the corresponding method on the object. Different scheduling policies are available and the user can attach priorities to messages. The advantage of this approach is that no chare can hold a processor idle while it is waiting for a message. Since $N>P$, there may be other chares that can run in the interim. The logical processes (LPs) of PDES (called

*posers* in POSE) are mapped onto CHARM++'s chares in a straightforward manner. Similarly, we use timestamps on messages as priorities and the CHARM++ scheduler serves as an event queue.

CHARM++ also provides generalized arrays of migratable parallel objects which allows us to implement our own load balancing strategies in POSE. Another benefit of using CHARM++ is its communication libraries[10]. POSE uses a streaming communication strategy to collect and periodically deliver messages by grouping them together into a single send operation. A last but not least reason for using CHARM++ to implement POSE is its highly portable nature and the existence of ports to most available parallel architectures and distributed environments.

### 3.1. Posers

*Posers* are CHARM++ chares representing entities in the simulation model. Each poser has a data field for *object virtual time* (OVT). This is the simulated time that has passed since the start of the simulation relative to the object. Posers have *event methods* which are CHARM++ *entry methods* that have a data field for *timestamp* in all messages sent to invoke them.

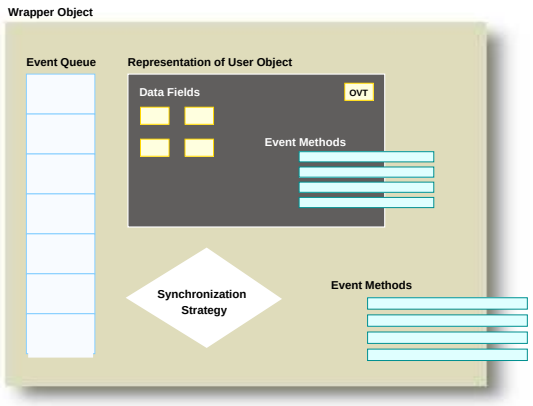


**Figure 2. Components of a *poser***

Posers have two methods for passing simulated time. The first is the *elapse* function. This is used in a poser when a certain amount of local time is passed (presumably performing some activity). It advances the OVT of the poser in which it is called. The second means is by an *offset* added to event invocations. This can be used as a means of scheduling a future activity or to indicate *transit time* in a simulation.

Posers have *plug-in behaviors* for their underlying implementation. These behaviors control the queuing of events, the synchronization of their execution and access to and modification of poser state. We refer to these respectively as the **wrapper** behavior, the **synchronization strategy** behavior, and the **representation** behavior. Different approaches or *plug-ins* can be used for each of these. Figure 2 illustrates how these components fit together. The simulation developer can concentrate on modeling entities (the representation). For more control over simulation behavior and performance, the developer can later look into trying different synchronization strategies.

Using virtualization allows us to maximize the degree of parallelism. Including an event queue in the object means that the scope of simulation activity resulting from a straggler is limited to the entity on which the straggler arrives. Since different entities may have dramatically different behaviors, we are also limiting the effects of those behaviors to a smaller scope. In particular, if one small data structure is a constantly updating part of a larger, more static entity, we want to separate it from the larger structure to avoid checkpointing the larger state. Further, encapsulating the relevant data in an object makes migration of that object much simpler.

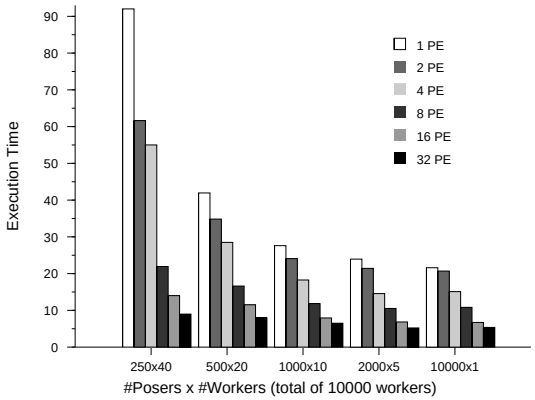


**Figure 3. Effects of virtualization**

Figure 3 shows the effects of virtualization. This is a simple simulation with 10000 `worker` entities. We organized the workers into `team`s. Each team is a poser object that encapsulates all the data associated with its workers. The simulation starts with each team giving each worker some work to do. Each worker performs its computation, then generates a few events for other workers. Some work is generated for a neighboring worker (likely to be on the same team) and some is generated for a distant worker. A random placement of posers on processors was used. We ran this simulation up to 32 processors for several team configurations. As the

figure shows, we achieved the best performance for the case with 10000 team posers, each with a single worker. It would seem that larger posers should have benefited from the fact that some of their communication was guaranteed to be local, but instead overhead from checkpointing, fossil collection and rollback dominated.

### 3.2. Speculative synchronization mechanisms

In POSE, an object gets control of a processor when it either receives an event or cancellation message via the scheduler, or when another component of the simulation (typically the GVT after a new estimate has been calculated) awakens the object. In the first case, the object's synchronization strategy is immediately invoked and in the second case, we perform fossil collection before invoking the strategy.

Our optimistic strategy checks for cancellation messages and handles as many as possible. Note that a cancellation may arrive before the corresponding event. Next, the strategy checks for any stragglers that may have arrived and rolls back to the earliest. Finally, it is ready to perform forward execution steps.

This is where the opportunity to perform speculative computation arises. All optimistic strategies perform some amount of speculative computation. In traditional approaches, an event arrives and is sorted into the event list and the earliest event is executed. The event is the earliest on the processor, but may not be the earliest in the simulation, so its execution is speculative. In our approach, we have a *speculative window* that governs how far into the future beyond the GVT estimate an object may proceed. Speculative windows are similar to the time windows[14] of other optimistic variants, except in how events within the window are processed.

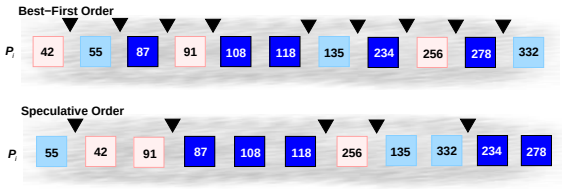


**Figure 4. Speculative event re-ordering**

Our strategy moves events from the CHARM++ scheduler to the event queues on the objects they are destined for in timestamp order. Each object gets a chance at forward execution for each event it gets and is allowed to speculate whenever it has control. If there are events with timestamp greater than or equal to the GVT estimate but within the speculative window, it executes *all* of them. The later events are probably not the earliest in the simulation and it is very likely that they are not even the earliest on that processor. We are allowing the strategy to speculate that those events are the earliest that the *object* will receive.

By handling events in bunches, we reduce scheduling and context switching overhead and benefit from a warmed cache, but risk additional rollback overhead. Figure 4 shows how events arriving on a processor might be reordered for better cache performance and fewer context switches (events of the same shade are intended for the same object). Table 1 compares the locality of reference for a simulation run sequentially versus the same simulation run with a speculative strategy. This shows that our speculative strategy is doing extra work, but benefiting dramatically from locality of reference.

**Table 1. Locality of reference comparison**

| | Sequential | Speculative |
|---|---|---|
| References | 4,597,178,671 | 8,536,026,179 |
| Cache Misses | 2,371,769,619 | 610,132,131 |
| Miss Rate | 0.515919 | 0.071477 |

The locality of reference benefit is exhibited in our performance analysis by a smaller average grainsize for the same events as compared to that of the sequential and best-first orderings of events. It is this smaller grainsize that we use to compute $T_I$ as described in Section 1.4.

### 3.3. Adaptive synchronization mechanisms

We have described the concepts of poser virtualization and speculative strategies as implemented in POSE and they have improved performance in new and novel ways. Next, we use these approaches to get at the heart of the general-purpose PDES problem.

We want POSE to perform well on any type of model. One model may differ in its behavior from another and the approaches described so far handle a wide variety of behaviors. However, a model itself may have different components that differ dramatically in behavior. In this case, the best way to speculate about which events to execute on one object may not be the best on another. To handle this situation, we have developed speculative synchronization mechanisms that can adapt to differing behaviors on a per object basis.

Consider two extreme cases. One poser receives many events from all over the system being simulated. It is very likely to receive stragglers that lead to rollbacks. It also has a large state that might be cumbersome to checkpoint, restore and commit. We would like to keep

a short leash on such an object to limit how much speculative computation it performs. Within the same simulation, we have another object with a small state. The object also receives a large number of events to execute, but the nature of the simulation results in those events arriving in order most of the time. For this type of object, we would like to allow it to speculate more than the first object, especially if there is idle time on its processor.

Our adaptive strategy takes some of these issues into consideration. It adjusts the speculative window size on a per object basis, according to the recent past rollback behavior of the object. The more successful speculative computation an object performs, the further into the future the object is allowed to speculate. Conversely, objects that receive stragglers are reined in and restricted in the amount of speculation they can do. This strategy effectively pushes our speedup curves for simulation closer to the near-linear goal discussed earlier.

Adding adaptivity at the object-level to a synchronization strategy collects data about object behavior that is useful for several other aspects of PDES. This information is useful for load balancing and communication optimization as well as GVT estimation. It has led to the development of adaptive load balancers, adaptive communication optimization frameworks and adaptive checkpointing strategies in POSE.

## 4. Performance of POSE

To better illuminate the problems with overhead in fine-grained simulations, we have designed a synthetic benchmark parameterized to exhibit the wide variety of behaviors found in PDES simulations. The benchmark creates objects placed on processors according to an initial distribution (uniform, random or imbalanced) and each object initially sends work to itself. A work event consists of performing computation for a time specified by a granularity (fine, medium, coarse, mixed, or a constant), then elapsing time according to a behavior pattern, spawning more work events for other objects.

We ran this benchmark with POSE using a parameter set with 5000 objects, high parallelism and fine granularity. The measured granularity was 0.00006s on average and the simulation executed about 81000 events. Figure 5 illustrates the overhead for this problem. This chart shows the average time per processor for each type of overhead in a simulation. The communication overhead represents all unaccounted for time between the maximum time spent in POSE on any processor and the simulation execution time. Thus it is a rough measure of time spent on communication and scheduling.

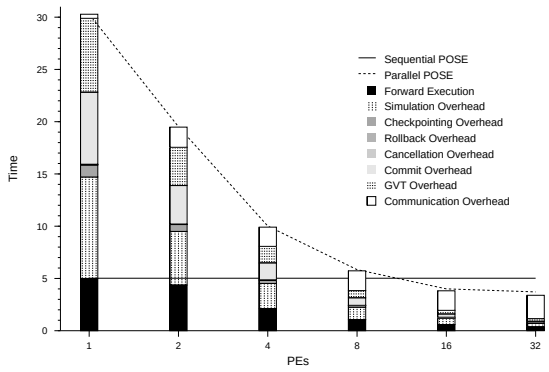


**Figure 5. Synthetic benchmark overhead**

The black section of the bars represents the time spent in the forward execution of events. The sequential time (shown as a solid horizontal line) is ideal sequential time $T_I$ and it corresponds to the single processor forward execution time. The next section of the bar represents all synchronization overhead. The grey sections are dominated by checkpointing (the darker) and fossil collection (the lighter) for this example. Above those we show the GVT overhead topped off by a white bar indicating communication overhead. For this simulation, we see that the break even point occurs between 8 and 16 processors. With all the overhead represented here, it is easy to see how hard it is to handle fine-grained events.

In spite of the small size of this example, we achieved near-linear speedups relative to single processor parallel time. Between 16 and 32 processors, the work was distributed too much to maintain a high degree of parallelism versus communication overhead.

### 4.1. Performance of synchronization strategies

To illustrate how these approaches perform in fine-grained simulations, we ran a small instance (5000 objects, high communication) of our synthetic benchmark with three synchronization strategies. The first is a simple optimistic approach with a time window, the second is speculative with a speculative window of the same size as the time window of the first strategy. The third approach is the adaptive strategy with a starting window of the same size as the other two strategies, but with freedom to adjust to the behavior of each object.

The simulation we ran had a very small grainsize averaging 23 $\mu s$ per event, with nearly 1,300,000 events executed. The comparative behavior of the three approaches appears in almost every experiment we run. Optimistic performs poorly compared to speculative ini-

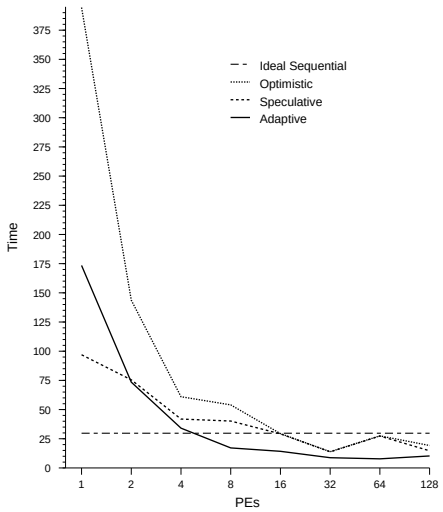


**Figure 6. Fine-grain benchmark run-time**

tially, but the difference dwindles as the number of processors increases. The adaptive approach has more overhead than speculative on fewer processors but eventually adaptive performs much better on higher numbers of processors. This behavior is shown in Figure 6.

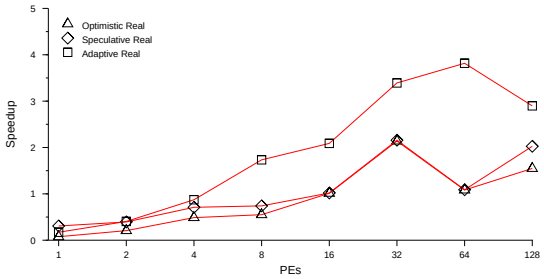


**Figure 7. Fine-grain benchmark speedup**

Speedup was difficult to achieve with such a small problem with small grainsize, but the adaptive strategy achieved a break even point at roughly 5 processors. The other strategies broke even at 16. The adaptive approach achieved a maximum speedup very near 4 on 64 processors as shown in Figure 7.

Contrast this example with the speedup for the same simulation executed with a larger grainsize of 262 $\mu s$ on average in Figure 8. The adaptive strategy achieves a speedup of 18 on 128 processors. The adaptive approach gets some of its gains as the number of processors increases because it is less susceptible to rollback overhead and achieves better locality of reference than the other strategies. However, another major reason for the performance improvement is that the speculative behavior allows the GVT algorithm to come up with higher estimates each time it is invoked, thereby requiring fewer GVT algorithm invocations than the other strategies.

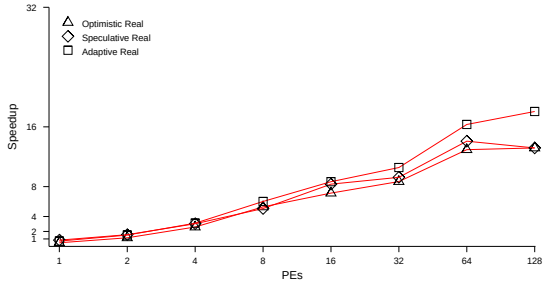


**Figure 8. Medium-grain benchmark**

## 4.2. Big machine simulation

BigSim[17] simulates performance of applications on very large parallel computers. It operates in two modes: an on-line mode which correctly predicts performance based on some preset message latencies, and an emulator-only mode which logs the tasks that were performed and their dependencies. In this second mode, the logged tasks have not been ordered in time with respect to each other and need to be timestamp-corrected in order to obtain performance results about the original program run on the BigSim emulator. We use POSE to perform timestamp correction. BigNetSim is an additional POSE module that plugs into the timestamp correction simulation and simulates the same application over a detailed network model. The behavior of the network model can be varied by its input parameters to model a variety of situations without ever needing to re-run the original program emulation. We discuss and present results for the timestamp correction phase of BigSim. The BigNetSim phase is in the preliminary performance analysis stage.

The timestamp correction simulation reads the trace log files generated by the BigSim emulator and creates posers to model the processors and nodes of the emulation. We start the first task at virtual time zero and let the tasks "execute" and record the virtual time at which each task starts, taking into account task dependencies and durations. We have an estimate of network latency which we use to determine how much time generated tasks spend in transit to the processor on which they will be executed. When all tasks have been executed, they will have correct timestamps and the final GVT should represent a correct runtime for the emulated application.

This simulation has very little computation taking

place in events and mostly involves the exchange and update of information. Thus it serves as a challenging fine-grained problem for POSE.

We used the BigSim emulator on 32 processors to run a 2D Jacobi program on 8000 simulated processors. We show a speedup plot for the correction from 1 to 64 processors in Figure 9. The simulator processed 5,085,836 events and had an average grainsize of 198 $\mu s$.

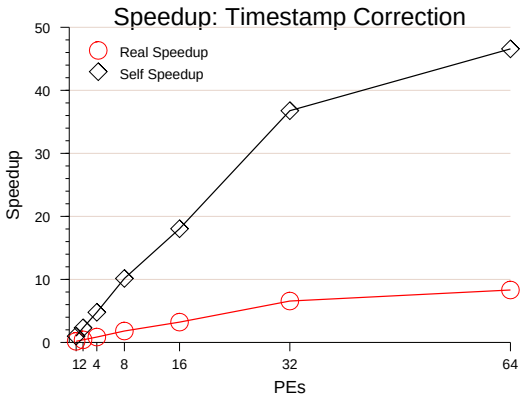


**Figure 9. Timestamp Correction Speedup**

The figure plots real speedup relative to ideal sequential time $T_I$ and self speedup relative to single processor POSE time. As the figure shows, self speedup is nearly linear to 32 processors, while real speedup shows a modest but correspondingly steady speedup improvement as we add processors. This problem did not achieve any speedup with traditional optimistic mechanisms.

## 5. Conclusions and Future Research

POSE has laid the foundation for a study of scalability in optimistically synchronized PDES. POSE incorporates the notion of virtualization from CHARM++, resulting in LPs being modeled by objects known as posers, several of which can be mapped to the same physical processor transparent to the user. The poser contains an instance of a synchronization strategy and its own event queue. This decentralization makes it possible to reduce the scope of overhead to just the objects directly affected. It also makes it possible for the synchronization strategy to react to the behavior of each object differently.

POSE expands the notion of speculation in optimistic synchronization. We have developed strategies that execute several fine-grained events on a single object with no intervening overhead. Not only does this reduce the per-event overhead, but it also has the effect of executing the events with a warmed cache and avoids the frequent context switching between objects that we would do if adhering to a strict timestamp ordering. Our strategy adapts to the behavior of each object by adjusting the speculative window, trying to obtain the largest set of events to execute speculatively on a single object while avoiding an increase in rollback overhead.

Future research will involve developing adaptive speculative strategies further to automatically react to simulation behavior. We will attempt to discern patterns in the forward execution behavior of objects and adjust speculation according to the pattern. Checkpointing frequency will also adapt to checkpoint only when state restoration is most likely to be needed. We are also developing improved strategies for communication in PDES. We use the priorities of messages to determine when they should be delivered. Finally, we have identified many factors affecting the load in a parallel discrete event simulation and are developing lightweight strategies for load balancing in POSE.

Ultimately, we hope to move the field of parallel discrete event simulation forward by developing techniques that allow parallel implementations of the most natural models to outperform their sequential implementations. Our results show much promise for improving the performance of general purpose PDES systems.

## References


[1] R. Bagrodia, R. Meyer, T. M, Y. Chen, X. Zeng, J. Martin, B. Park, and H. Song. Parsec: A parallel simulation environment for complex systems. *Computer*, 31(10):77–85, October 1998.

[2] R. L. Bagrodia. Perils and pitfalls of parallel discrete event simulation. In *Winter Simulation Conference*, 1996.

[3] R. L. Bagrodia and W.-T. Liao. Maisie: A language for the design of effi cient discrete-event simulation. *IEEE Transactions on Software Engineering*, 20(4):225–237, April 1994.

[4] S. R. Das, R. Fujimoto, K. S. Panesar, D. Allison, and M. Hybinette. GTW: a time warp system for shared memory multiprocessors. In *Winter Simulation Conference*, pages 1332–1339, 1994.

[5] R. Fujimoto. Parallel Discrete Event Simulation: Will the Field Survive? *ORSA Journal on Computing (feature article)*, pages 213–230, 1993.

[6] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[7] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto. Time warp operating system. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 77–93. ACM Press, 1987.

[8] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[9] L. V. Kalé. The virtualization approach to parallel programming: Runtime optimizations and the state of art. In *Los Alamos Computer Science Institute Annual Symposium*, October 2002.

[10] L. V. Kale, S. Kumar, and K. Vardarajan. A Framework for Collective Personalized Communication. In *Proceedings of IPDPS'03*, Nice, France, April 2003.

[11] Y.-B. LIN and E. LAZOWSKA. Determining the global virtual time in a distributed simulation. In *Proceedings of the International Conference on Parallel Processing*, pages 201–209, August 1990.

[12] F. Mattern. Effi cient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.

[13] B. R. Preiss. Performance of discrete event simulation on a multiprocessor using optimistic and conservative synchronization. In *International Conference on Parallel Processing*, 1990.

[14] L. M. Sokol, J. B. Weissman, and P. A. Mutchler. Mtw: an empirical performance study. In *Proceedings of the 23rd conference on Winter simulation*, pages 557–563. IEEE Computer Society, 1991.

[15] J. S. Steinman. Breathing time warp. In *Proceedings of the seventh workshop on Parallel and distributed simulation*, pages 109–118. ACM Press, 1993.

[16] A. I. Tomlinson and V. K. Garg. An algorithm for minimally latent global virtual time. In *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation*, pages 35–42. ACM Press, 1993.

[17] G. Zheng, T. Wilmarth, O. S. Lawlor, L. V. Kalé, S. Adve, and D. Padua. Performance modeling and programming environments for petaflops computers and the blue gene machine. In *NSF Next Generation Systems Program Workshop, 18th International Parallel and Distributed Processing Symposium(IPDPS)*, page 197, Santa Fe, New Mexico, April 2004. IEEE Press.