POSE: SCALABLE GENERAL-PURPOSE PARALLEL DISCRETE EVENT SIMULATION

BY

TERRY LYNN WILMARTH

B.S., State University of New York at Buffalo, 1991
M.S., State University of New York at Buffalo, 1993

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

Abstract

Parallel discrete event simulation (PDES) applications encompass a broad range of analytical simulations. Their utility lies in their ability to model a system under study and provide information about the behavior of that system in a timely manner. The most comprehensive models for such systems can be vastly complex, have highly irregular structures and fine granularity, making them challenging problems to parallelize. Current PDES methods provide limited performance improvements over sequential simulation and complicate the modelling process, requiring knowledge of specialized parallel computing practices that may be well outside the application developer's field.

We propose a novel environment for PDES that facilitates the development of highly parallel models and requires minimal understanding of parallel computing concepts. We propose four primary approaches to improving the performance of PDES. We first examine the overhead required for synchronizing events to obtain correct results in parallel and develop a new approach to the structure of model entities and mechanisms for PDES that help to reduce that overhead. Secondly, we design new adaptive synchronization strategies that exploit this new model structure to obtain better cache performance and reduce context switching overhead. We then develop techniques to optimize communication in concert with these new strategies. Finally, we study load balancing in the context of optimistic synchronization and design new approaches to fit with our other techniques. These four approaches form an integrated system for handling non-ideal simulation models. We demonstrate our techniques via a highly flexible synthetic benchmark capable of mimicking a variety of simulation behaviors, as well as with simulations of network models for very large parallel computers.

In loving memory of my father

Harold “Jerry” Wilmarth

1936–2001

and my grandmother

Frances H. DiPietro

1912–2001

Acknowledgements

This work would not have been possible without the patience and persistent insistence of my advisor Professor L. V. Kalé. His invaluable ability to warp time, somehow creating time for spontaneous thirty-minute meetings when he only had ten minutes to spare, was greatly appreciated. I would also like to thank my dissertation committee, Prof. Michael Heath, Prof. Sarita Adve and Prof. Placid Ferreira, for their very helpful suggestions and advice. In addition, Dr. David Jefferson's feedback on this work has been extremely helpful.

Over the years, several members of the Parallel Programming Laboratory have been essential to the completion of this work. Some of these people provided their assistance in the form of technical wizardry, while others made it possible to tolerate the daily grind. Indeed, some of them may have been the reason it took so long. I'd like to thank, in no particular order, Chee Wai Lee, Sayantan Chakravorty, Vikas Mehta, Yogesh Mehta, David Kunzman, Nilesh Choudhury, Filippo Gioachin, Gengbin Zheng, Orion Lawlor, Sameer Kumar, Praveen Kumar Jagadishprasad, Joshua Unger, Milind Bhandarkar, Joshua Yelon, Jay DeSouza, Chao Huang and Robert Brunner.

To my family and friends a huge heartfelt thank you. Thanks for understanding that I believe life is for living, not for putting on hold to "do important things". Special thanks to Lee Baugh (beer, hikes, guitar, music, books, movies, SF walks, etc.) and Chris Cheung, for always being an instant message away. Thanks to Kulbir Arora for making Computer Science fascinating and teaching me to think outside the black box. Thanks to my wonderful family; I hope to see you all a lot more often after this is over! Much love to you all: my brothers, Steve Wilmarth and Andy Wilmarth, my uncle Peter DiPietro, my aunt Pat DiPietro, my

"sisters" Carol Lindsay and Barbara Weise, my "nieces" Heidi and Jennifer.

Thanks to Dervish and Fiona, my "kiddies", for always knowing when I needed a cuddle and for all the amusing "laptop contention" incidents.

My father memorized the title of my thesis and had a basic understanding of what it was all about. He then went bragging to his buddies at work about his daughter and her thesis topic. Thanks Dad, for always being proud of me and believing I could do more. I miss you terribly. I will walk the high places and the unworn paths and see all the trees for you. You'll be right there with me.

My grandmother was all things good to me. A light, a hope, a comfort in the darkest times of my youth. Thanks for everything.

Most of all: Thanks Mom. My mom, Sandra Wilmarth, is the reason this went on and did not stop somewhere short. I hope I someday learn to tap into her source of infinite strength. Words fail to express my gratitude and love.

And... through it all, one person has put up with the whining, the moping, the rare but violent beatings on the keyboard, and the even rarer frantic jumps-for-joy at some success however small. My sweetheart, Eric Bohm, has been my source of inspiration, has looked over my shoulder as I sought bugs, has hacked code, proofread innumerable papers and drafts, performed code reviews, been a Pose victim, nodded and smiled at all my wacky ideas, made countless pots of coffee, cooked numerous meals and just generally soothed the ache that is Ph.D. research. He has helped me through triumph and tragedy and I am forever grateful.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Discrete event simulation (DES) applications are used to model and analyze a broad range of domains and virtual environments. Their utility lies in their ability to model the system under study and provide information about the behavior of that system in a timely manner. However, as the size and complexity of simulations grows, it becomes less feasible to use sequential simulation. Parallel discrete event simulation (PDES) addresses the space and time limitations of sequential DES. Current methods for PDES provide limited performance improvements over sequential simulation because the most comprehensive models for these applications can be vastly complex, have highly irregular structures and fine granularity, making them challenging problems to parallelize. In addition, the conceptual overhead of modeling systems to utilize these methods is large, requiring knowledge of specialized parallel computing practices that may be well outside the application developer's field. The most promising results are typically obtained by *developers* of PDES systems, while simulation models developed by *users* of such systems perform poorly.

We propose a novel environment for PDES that facilitates the development of highly parallel models and requires minimal understanding of parallel computing concepts. This dissertation is aimed at overcoming performance problems due to fine granularity in optimistic synchronization of events and exploiting any parallelism in a model as much as possible.

We focus our efforts on four primary approaches to improving the performance of PDES:

- We first examine the overhead required for optimistically synchronizing events to obtain correct results in parallel. From this study, we develop a new structure for simulation model entities. This structure is designed to reduce *per object* and per event overhead and ensure that remaining overhead scales well. As we add processors, the time spent on overhead should reduce, achieving linear (or better) speedups for overhead components. Similarly, if we double the number of events performed in a simulation, the time to execute them should at most double. The approaches we use are frequently capable of superlinear speedups for certain types of overhead and often handle an increase in events executed with little or no increase in cost.

- Secondly, we design a new adaptive synchronization strategy that exploits this new entity structure to adapt the forward execution behavior of each entity separately to obtain a better overall speedup. This strategy allows a simulation to automatically tune itself to the changing situations that it may encounter over the course of a program execution. It also makes the simulation more adept at achieving optimal performance in changing computational environments such as differing number of processors, processor speeds, architecture, network, etc. In some situations, the strategy exhibits better cache performance and reduces context switching overhead over traditional optimistic synchronization mechanisms.

- We then develop techniques to optimize communication in concert with these new strategies. We have explored direct streaming and 2D mesh-based streaming strategies designed to intelligently group small messages into single send operations to reduce the dominant per-message overhead. We have also incorporated the timestamps on events into our communication optimizations, treating the earliest timestamped events as high priority and using special streaming strategies to expedite the delivery of these high priority events.

- Finally, we study load balancing in the context of optimistic synchronization and design new approaches to fit with our use of adaptive speculation in synchronization. These approaches acknowledge the possible absence of idle time in imbalanced simulations and define other measures of load and balance. They arrange work such that all processors have a balance of useful work (i.e. work that is not likely to be rolled back) over time.

These four approaches form an integrated system for handling challenging simulation models.

1.1 Motivation

What is a computer simulation? "A *computer simulation* is a computation that models some real or imagined system over time[13]." Why is computer simulation necessary? Computer simulation makes it possible to analyze systems that would be expensive, dangerous or even impossible to construct prototypes for. This research focuses on *discrete event simulations* (DES) in which the simulation is driven by events that can occur at discrete points in time and in different parts of the system being modelled, as opposed to lock-step simulations with continuously occurring behaviors that affect the entire system. We describe sequential and parallel DES in detail in Chapter 2.

In this thesis, we study analytical simulations which are used for modeling complex systems to obtain a deeper understanding of that system. For example, we might study the evolution of a population of bacteria in an environment that is hard to reproduce in a laboratory by modeling the expected behavior and observing the results. We can design complex systems to model industrial or other processes and weed out faults in the system and optimize it for the best performance before the system is ever built. Simulation can also be used to handle special situations where decisions need to be made quickly. For example, in foul weather, an air traffic controller needs to quickly determine how to reroute incoming airplanes, a decision that must be made in minutes or even seconds rather than days or hours. We also use simulation for discovery: we can develop and simulate mathematical models to test our theories about the world, compare the results to observed behaviors and learn from the deviations how to improve our model.

Simulations of some physical systems are extremely complex; computers are necessary to visualize and analyze massive quantities of data. However, they are also too complex for simulation on a single computer due to limitations on the memory size of a single processor and excessive execution time. We must find ways to utilize the available parallel and distributed computing power to execute these simulations efficiently. Consequently, we arrive at the need to parallelize our simulations.

Parallelism should have the obvious result of reducing the execution time of simulations. Sequential simulation also cannot handle extremely large-scale simulations since the active entities in the system may not be able to fit into memory on a single machine. Parallel simulation not only makes huge simulations feasible, but can also potentially achieve better cache performance when we partition the entities across the parallel machine since there are fewer entities per processor.

When we provide a means for parallel simulation, we must also remember that simulation is a multidisciplinary tool. The model developer is not necessarily a computer scientist. The systems being simulated are already highly complex, so the tools used to simulate them should not complicate them further.

Limited progress has been made in PDES research over the last two decades. Large speedups have been obtained for a very few specific application domains with more modest speedups for a handful of other problems. However, some major difficulties remain as discussed in an article by Fujimoto[11]. Summarizing a table in this article, speedups range from 5 on 54 processors to 1900 on 16,384[1]. Efficiencies of these runs range from as little as 8% to 93% with an average of 50%. Fujimoto points out that these results are misleading: they are compiled from the best published results and some are from highly specialized simulation applications. The figures also leave out details pertaining to the effort that went

[1] This simulation modelled *Ising spin* and used the conservative bounded lag protocol

into obtaining such results. Further, those results were almost always obtained by highly trained experts in the field — usually the same person who developed the underlying parallel simulation environment. Simulations developed by non-experts tend to perform poorly. There is a strong need to develop parallel simulation environments that can perform well when an application domain expert is developing the code. Related to this is the problem that PDES involves a certain amount of overhead to accomplish synchronization and cannot perform well when the size of the units of simulation computation (*i.e.* the *granularity* or *grainsize*) is small. This flies in the face of allowing the simulation designer to use the most logical breakdown of the system into its components, since many simulations naturally have fine granularity. To quote Fujimoto on the subject, "a simulation model that is developed according to what is most 'natural' or 'elegant' to the simulation programmer is often one that is poorly suited for execution on a parallel computer, even if the system being simulated contains substantial amounts of concurrent activity[11]." In this thesis, we strive to address these issues by designing techniques to handle the adverse situations that arise in application models.

As more parallel computing power rapidly becomes available, we wish to use that power along with PDES to simulate more complex applications. We hope our efforts improve the performance results over those seen in the current literature and that similar results are obtainable by anyone using our parallel discrete event simulation (PDES) mechanism.

1.1.1 An Example of the Challenges

Parallel machines with enormous computational power and scale are now being built consisting of tens of thousands of processors and capable of achieving hundreds of teraflops of peak speed. For example, the BlueGene (BG/L) machine being developed by IBM will have 128K processors and 360 teraflops peak performance. Ambitious projects in computational modeling for science and engineering are preparing to utilize this power to achieve breakthroughs in areas such as rational drug design, genomics, computational astronomy and engineering design.

We have used PDES to predict the performance of applications such as molecular dynamics on such machines, from modelling programming environments to detailed network contention simulation. Our experience with simulating network contention models illustrates much of what is difficult about parallel discrete event simulation. Modelling processors, nodes, switches and the interconnects between them requires a significantly large state, while routing algorithms and topologies determine what little computation is performed on each entity. The grainsize of computation on the entities is very small, while the state represented is large. In addition, while artificially generated network traffic can keep a simulation busy, simulations with traffic generated by real applications suffer from a low degree of parallelism. We discuss this case study, the BigNetSim project, further in Section 11.

1.2 Research Objectives and Contributions

This thesis explores new techniques for tolerating fine event granularity and achieving scalability in optimistically synchronized PDES. The primary research objectives are to reduce overhead, improve parallelism, achieve latency tolerance and improve human productivity through a variety of mechanisms:

- **Overhead Reduction, Improved Parallelism and Latency Tolerance via Virtualized Object Model:** We achieve various performance improvements through a well-designed object model that relocates the function of the PDES concept of “logical process” (LP) to individual objects. This results in overhead reduction since the scope of overhead is restricted to operations on the object, and the behavior of individual objects determines how much overhead will be encountered. Parallelism and latency tolerance are achieved through virtualization. There should be many such objects per physical processor; if one object is waiting for work, it is likely that another object will have work that can be done in the interim.

- **Overhead Reduction and Improved Cache Performance via Adaptive Speculative Synchronization of Events:** We allow events on a physical processor to be executed out of timestamp order in some cases as long as they are executed in order on the individual objects. We allow objects to speculate that the work available to them will not need to be rolled back and thus they can execute all the work together, thereby improving cache performance and reducing context switching overhead. An adaptive synchronization strategy determines how much an object speculates based on its past behavior, present state and pending future events. This should maximize the caching effect and context-switch reduction while avoiding excessive rollback overhead due to over-speculation.

- **Improved Parallelism via Asynchronous Global Virtual Time Approximation:** We develop a global virtual time (GVT) approximation algorithm that runs simultaneously with the simulation to avoid the idle time of synchronous approaches. Other asynchronous approaches exist, but ours is unique in that it is invoked in a decentralized manner, requires only a single phase of information gathering to obtain a new estimate, and runs during otherwise idle periods of time on processors. Our approach requires less overhead and network traffic than other multi-phase asynchronous strategies.

- **Latency Tolerance via Communication Optimization:** We employ various techniques to group small event messages into a single buffer and transmit them with a single send operation. This saves considerable time spent on per message transmission cost. We also explore techniques to expedite the delivery of high priority (*i.e.* low timestamp) events in this context.

- **Improved Parallelism and Reduced Overhead via Load Balancing:** Load balancing optimistically synchronized PDES applications presents us with challenges that differ from those of traditional applications. For example, we can no longer rely on idle time on a processor as a measure that a processor is underloaded. When speculation is allowed, there may never be any idle time. Load balancing must be based on improving the mix of high, medium and low priority work available on each processor.

- **Improved Human Productivity:** How difficult it is for the simulation developer to specify the model and implement the parallel simulation depends on the simulation language. Many parallel discrete event simulation languages have a high knowledge overhead: they require the developer to thoroughly understand parallel computing as well as the underlying simulation language implementation [11].

This thesis makes several contributions to the study of PDES. It develops novel approaches to optimistically synchronized PDES that are in some ways non-obvious and apparently counter-intuitive to the objectives of the thesis, yet they achieve the desired effects and inspire a new way of thinking about PDES. Another main contribution is the development and implementation of a PDES environment, Pose [49], that incorporates these new approaches. Pose has so far been used in the development of detailed contention-based network models for large parallel machines such as BlueGene/L. We include an in-depth performance study involving both synthetic and real benchmarks with performance measurement on a variety of parallel machines and clusters. The thesis reveals that the challenges of PDES may not be as insurmountable as once thought and that further research is merited.

1.3 Related Work

Several surveys provide an overview of the challenges and progress made over the years in PDES[10, 12, 11]. Since our focus is on optimistic synchronization protocols, we examine several such approaches. One of the most well-studied optimistic mechanism[34] is Time Warp, as used in the Time Warp Operating System[16]. Time Warp was notable as it was designed to use process rollback as the primary means of synchronization.

The Georgia Tech Time Warp (GTW)[8] system 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. More recent versions exist for networks of homogeneous workstations. The designers of GTW recognized that overheads in event processing had to be dealt with efficiently to achieve better performance with such granularities. One of the GTW features that carries over into Pose is the *simulated time barrier*, a limit on the time into the future that LPs are allowed to execute. Pose uses a similar concept in its *speculative window*. GTW achieved speedups as high as 38 using 42 processors for simulating a personal communication services (PCS) network.

The Synchronous Parallel Environment for Emulation and Discrete Event Simulation (SPEEDES)[43, 44] was developed with a different optimistic approach to synchronization called *breathing time buckets*. SPEEDES has since evolved to include a number of strategies for comparison purposes, including versions of Time Warp. SPEEDES has continually evolved over the years and has been used for a variety of applications, including military simulations.

There have been many PDES systems that allow for combinations and/or hybrids of various existing synchronization mechanisms. One such early system was Yaddes[40, 39]. The Yaddes system allowed a program to be run sequentially, or in parallel using one of three mechanisms (multiple event lists, conservative, and optimistic) with no changes. Maisie[4, 2] is another example of such a system. Minor modifications can render a Maisie program executable as a sequential simulation, or in parallel using either a conservative or an optimistic synchronization mechanism. MOOSE[48] is an object-oriented version of Maisie that supports similar synchronization abilities. Maisie's successor, Parsec[1], also allows for a mix of synchronization protocols amongst the LPs of a single program. The best speedup reported for Parsec in 1998 was very near 8 on 16 processors for simulating a 3000-node wireless network, using the conservative null message protocol, and the GloMoSim library for simulating such networks in Parsec. Research into adaptive protocols has been conducted as well [7] focusing on the belief that the least costly solution involves a trade-off between the conservative and optimistic extremes. Our research on adaptive strategies is unique in that it can still be considered aggressively optimistic: our adaptive strategy always tries to increase the risky work that it performs (and performs much higher risk work as well), and no mechanisms are employed to avoid risk (rollbacks). When we perform risky events, we perform them the same way as any other event; we do not hold back events that are generated and we do not perform state modifications on a temporary copy of the state. Because of the low-cost of optimistic synchronization brought about by our object model (Chapter 5), we only need to apply a throttling mechanism when rollbacks get dramatically out of hand.

The challenges of low parallelism in PDES have been addressed by algorithms which spread the computation and data of single entities across multiple processors[14]. This approach abandons the use of distributed event queues and poses the simulation problem instead with recurrence relations. The approach, however, is not always applicable. Our approach, by contrast, simply suggests that objects be broken apart into smaller entities. This makes it a general-purpose simulator for any discrete event system.

Load balancing in the context of optimistic synchronization has been studied extensively in the past. The need to study new approaches for load balancing in optimistic PDES was clarified early on[41]. Reiher and Jefferson acknowledged the need to use new metrics for determining load, and thus they used *effective utilization* to express the fraction of non-speculative work on a processor. Their work on TWOS improved performance by 13-19%. Wilson and Nicol studied automated load balancing in SPEEDES[50, 51, 52]. Another approach uses learning algorithms and flow control to achieve load balance in Time Warp[6]. Still others approach specific types of PDES problems and develop load balancing techniques for particular situations [9].

1.4 Thesis Organization

In Chapter 2, we give a brief introduction to parallel discrete event simulation and our motivations to study the optimistic method for event synchronization. In the next chapter, we discuss the Charm++ language as a backdrop for our PDES studies. Following this, we describe our performance goals for PDES and define the terminology we will be using throughout the thesis. This chapter also details the main obstacles to scalability in PDES and describes our synthetic benchmark which exhibits many of these behaviors. The impact of virtualization and the Pose object model are discussed in Chapter 5. We study the behavior of the new object model through a number of experiments. Chapter 6 discusses new approaches to optimistic synchronization enabled by our object model of the previous chapter and analyzes the performance of these strategies. In Chapter 7, we discuss our asynchronous algorithm for global virtual time (GVT) estimation. Communication optimization and load balancing for PDES are described in detail in Chapters 8 and 9. We study the improvements to performance enabled by these methods in detail. Chapter 10 discusses techniques for post-mortem analysis of simulations. The next chapters illustrate the usage of Pose in practice via a case study. Finally, we consider future explorations with Pose in Chapter 12 and present our conclusions in Chapter 13.

Chapter 2

An Introduction to Parallel Discrete Event Simulation

Parallel discrete event simulation is a fascinating field of study attracting much attention from researchers. In his classic survey article, Richard Fujimoto [12] suggested two perspectives to that attention: the pragmatic view of the field is that large simulations in a variety of disciplines consume huge amounts of time when run sequentially, and the academic view is of a problem domain with a high degree of parallelism which in practice is paradoxically difficult to parallelize. He goes so far as to suggest that a "sufficiently general solution to the PDES problem may lead to new insights in parallel computation as a whole".

Before we can describe PDES, it is helpful to go over how discrete event simulation works and how it is applied. In short, each system has a *state* (of its data) associated with some point in time. If the entities that comprise the state can only change at discrete points in time, then we can use discrete event simulation to model it. These state changes are known as *events* and the point in time at which the event can happen is its *timestamp*. When we talk about the time that passes in the simulated world, we will frequently distinguish it from real time or execution time by calling it *virtual time*.

It should be noted that discrete event simulations have events that may occur at irregular jumps in simulated time, as opposed to lockstep simulations. In a lockstep simulation, *all* the component entities handle events at regular intervals. It helps to think of discrete event simulation as a means to exploit the *redundancy* in the system, *i.e.* the fact that for periods of time, portions of the state may not change. Thus lockstep simulations of such systems are very inefficient. Of course, the fact that we wish to execute events with differing timestamps on the state *concurrently* should raise a red flag. Clearly, this will lead to complex synchronization problems.

A discrete event simulation has three main data components as shown in Figure 2.1: the *state* which represents the physical entities in the system being simulated; the *event list*, a queue of events to be performed on the state; and a discrete *global clock* which is the point in virtual time to which a simulation has progressed. Each event has a timestamp which is the point in virtual time at which the event is available to be executed. The 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 many of the events in the list.

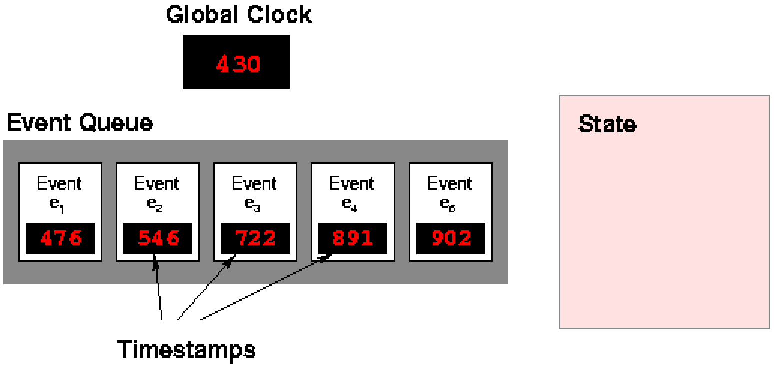


Figure 2.1: Data components of discrete event simulation

To run a simulation in parallel, we would like to be able to process events concurrently. One way to do this would be to have separate event queues on concurrent *logical processors* (LPs) that access a shared state as shown in Figure 2.2. This arrangement can result in *causality errors*. If we execute two events at differing timestamps concurrently and the event with the later timestamp modifies the state before the event with the earlier timestamp accesses it, the future will have changed the past! For example, consider the case of a traffic simulation with a traffic light. Suppose event e_2 is "the light turns from red to green" and that event e_1 is "Sally checks the light and proceeds accordingly". Allowing these two events to occur out of order could have disastrous consequences for our simulation (and for Sally). This problem forces us to avoid using a shared state in PDES. Thus, we must partition the state amongst the concurrent processes, which works well in a distributed memory environment. Events that are to occur on particular portions of the state are executed by the LP associated with that portion.

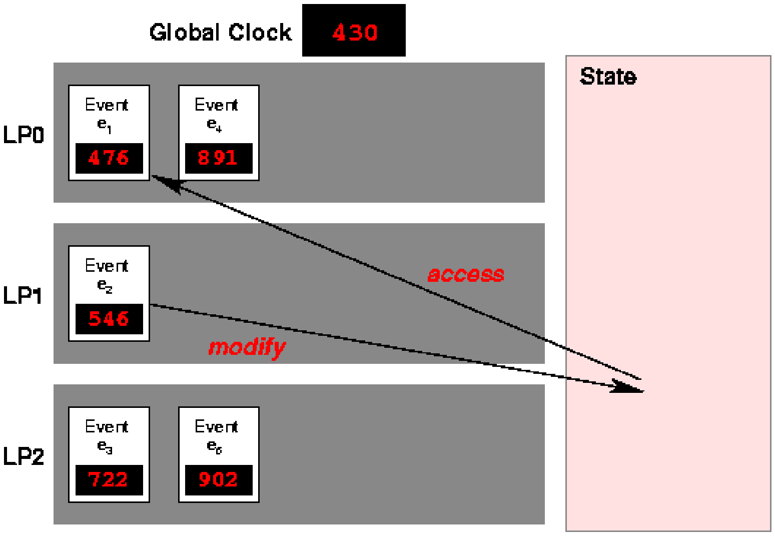


Figure 2.2: Shared state incurs causality errors

Partitioning the state does not eliminate all problems. Much of the work in PDES is devoted to adhering to (or working around) *sequencing constraints*. Sequencing constraints arise when a causality relationship crosses the boundary of an LP.

For example, consider two LPs with two events e_2 and e_5, one on each LP. Let $T(e)$ be the timestamp of an event e. For our example, $T(e_2) < T(e_5)$, but there is no other relationship between e_2 and e_5. It would seem that it should be safe to execute the two events concurrently. Suppose however that e_2 issues another event e_3 which is sent to LP_2, and $T(e_3) < T(e_5)$. Then e_3 must execute before e_5 on LP_2 because it may modify the local state which is later accessed by e_5. If we were to execute e_2 and e_5 concurrently, a causality error would have resulted. This situation is illustrated in Figure 2.3.

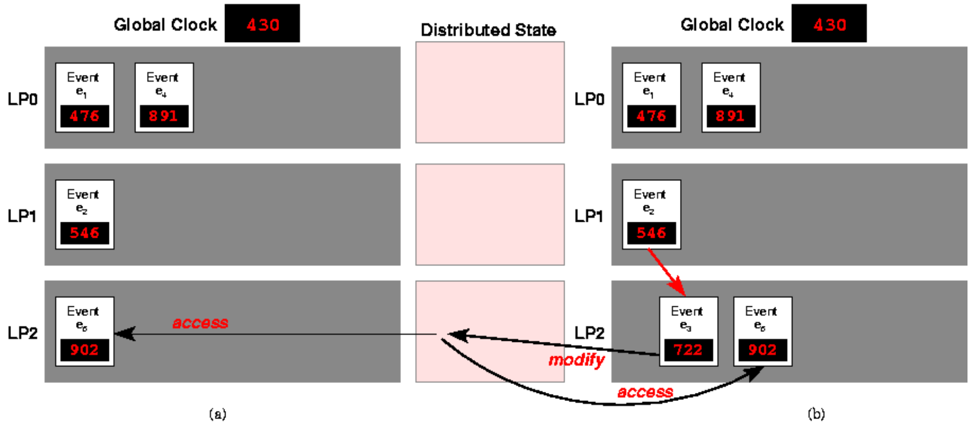


Figure 2.3: (a) No apparent obstacles to executing e_2 and e_5 concurrently; (b) However, e_2 generates e_3 resulting in a causality error

2.1 What Makes PDES So Difficult?

We've just illustrated the issues of causality and sequencing that make PDES such a challenging problem. An LP, on its own, cannot know if the event it has with the earliest timestamp is *the* earliest event it will have. Since PDES problems tend to be highly irregular with strong data dependence, solving these constraints to determine the order of event execution is a complex and dynamic problem for which a general solution is elusive.

Methods for synchronizing the execution of events across LPs are thus necessary for assuring the correctness of the simulation. There are two broad categories of mechanisms to do this in PDES. We discuss these in the next two sections.

2.2 Conservative Synchronization

The *conservative* approach avoids causality errors by determining the safety of processing the earliest event on an LP. An event is considered safe to process only when no other event could affect it (by generating events with lower timestamp on the same LP, for example). Determining safety is a complicated process that could lead to deadlocks, so deadlock avoidance or deadlock detection and recovery must be employed. In the worst case, this approach can completely serialize the simulation. To avoid such degrading to the sequential case, it is also helpful if LPs have some *lookahead* ability, so that they can predict that events earlier than a certain time cannot be generated. Because an unsafe earliest event causes an LP to block, conservative methods are often limited in the degree to which they can utilize the available parallelism[1]. Note also that an unsafe event *may* lead to a causality error, but there is also a chance that it may *not*. Thus we are postponing available work until we are sure about its safety, reducing the potential parallelism in the program.

2.3 Optimistic Synchronization

Alternatively, the *optimistic* approach to synchronization allows LPs to process the earliest available event with no regard to safety. When causality errors do occur, they are detected and handled. Detection happens when an event arrives for an LP with a timestamp that is less than that of some event(s) already processed. Such an event is known as a *straggler*. A *rollback* mechanism is then employed to undo the execution backward in virtual time to the point where the straggler should have been executed. Thus we see that some means of recreating the history of events executed on an LP is necessary. The state at earlier points in time must be recovered, either by periodic *checkpointing* of the state by storing it in memory, or by some form of *anti-methods* which can precisely undo an event method's changes to the state. Figure 2.4 illustrates the many activities involved in optimistic synchronization.

As we've seen, the optimistic approach allows the earliest event on an LP to execute without making any checks for the safety of that event. When an event is executed, there are two ways in which it can affect the future of the simulation. First, it can change the local state, and second, it can spawn other events. When a straggler arrives, we must handle these two effects for each event that we must roll back in order to reach a simulated time

[1] Some applications could easily be strong exceptions to this and are more amenable to conservative synchronization. These applications would likely exhibit more regular event behavior with ample parallelism at each individual timestep, approaching a nearly continuously changing state over all entities similar to a lockstep simulation.

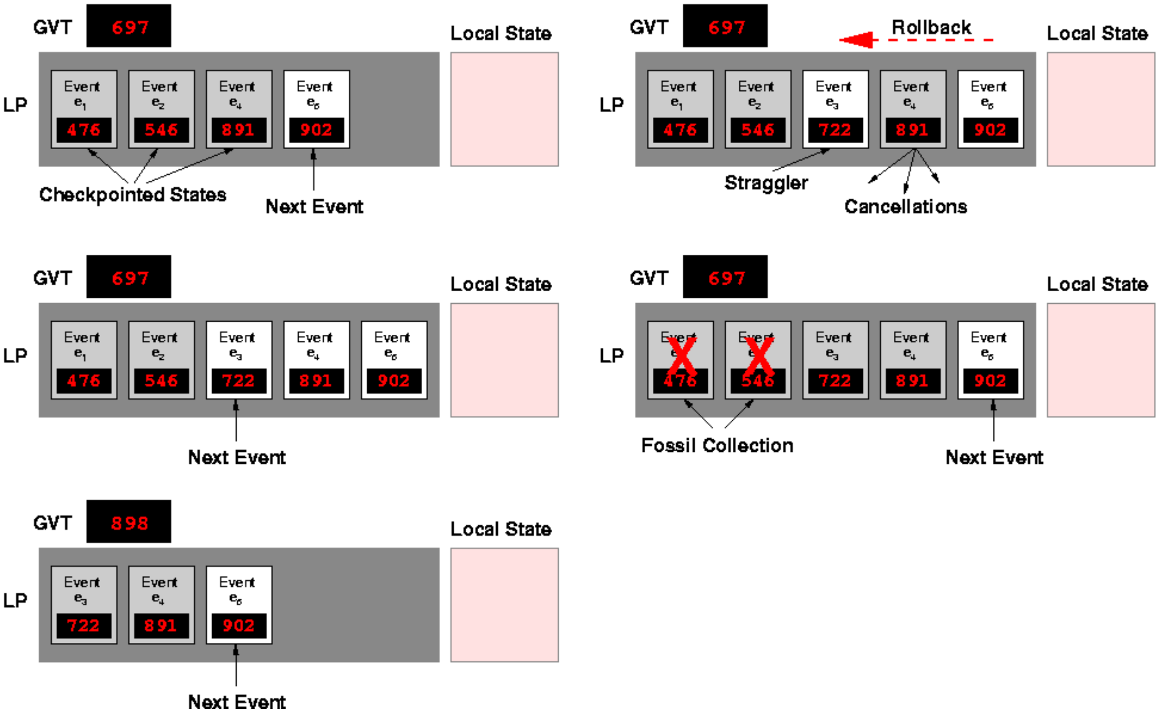


Figure 2.4: Activities performed in optimistic synchronization

prior to the straggler.

One way to handle the recovery of the state is to periodically checkpoint the state during *forward execution*. Then when we rollback, we simply restore the state at the appropriate point before the straggler's timestamp. Another way we propose for handling this is to provide or derive *anti-methods* for each event, that are capable of restoring only the portions of the state that were changed by the event. Anti-methods are not always feasible when operations on the state destroy its contents in an unrecoverable way (for example, `A = A+1` *vs.* `A = 5`). Such methods are difficult to derive using compiler support and therefore require effort on the part of the programmer designing the simulation. Anti-methods have several advantages over checkpointing. They significantly reduce the forward path overhead as well as memory usage. They can even reduce rollback overhead when the modifications they perform are minimal. One direction that could be taken is to examine ways to combine anti-methods with checkpointing to reduce the size of the checkpointed state.

To handle the spawned events, we send out *cancellation messages* (also called *antimes-*

sages which are not to be confused with anti-methods). Cancellation messages cause the removal of events from an LP's event list. In the case where the event to be cancelled has already been executed, further rollbacks are required. It is easy to see how this might cause a *cascade* of rollbacks throughout the simulation.

Another requirement in optimistic mechanisms is a way to free the memory used by checkpointed states. We can only do this when we are certain the checkpoints are no longer needed. Along with this goes the problem of producing output or collecting statistics in an event when we do not know if that event will *remain* executed. Printing information to standard output, for example, is impossible to rollback should the event be cancelled. To assist in these problems, optimistic simulations periodically calculate a *global virtual time* (GVT): this is the earliest timestamp of all unexecuted events or potential events in the simulation at any given point in time. We know that nothing earlier than the GVT can be rolled back or cancelled, so it is safe to dispose of any checkpoints with a timestamp earlier than the GVT. The process of reclaiming this memory and *committing* the event to history (thereby performing the I/O and other tasks) is called *fossil collection*, and in this thesis this is what we mean when we refer to *committing* events.

One of the most well-known optimistic protocols is the Time Warp mechanism[16]. There are many variations in and optimizations to how Time Warp has been implemented over the years. There are numerous algorithms for calculating the GVT, techniques for lazy cancellation[34] of events (when re-execution of an event after a rollback would not affect what new events get spawned, we do not cancel them), direct cancellation (gives cancellation message higher priority to reduce the chances for cascades of rollbacks) and lazy re-evaluation (when the straggler or cancelled message had no effect on the state, we need not re-evaluate each event when we re-execute), and various types of "time windows"(used to prevent propagation of errors too far into the future of a simulation) to name a few.

Besides conservative and optimistic synchronization mechanisms, there are also mixed or *hybrid* approaches which allow both mechanisms to be used. Some approaches can be *adaptive* as well – the simulation uses an appropriate synchronization strategy to handle the current situation.

2.3.1 The Case for Optimistic Synchronization

We feel that optimistic mechanisms have greater potential for utilizing available parallelism than conservative mechanisms. Optimistic approaches perform what we term *speculative computation*; computation that we hope will be useful and will not need to be rolled back or cancelled. In game-tree style search problems, we perform speculative computation when we explore several branches of a search space in parallel in the hope that one of them will lead to a solution. We prioritize the branches of the search space by some heuristic that predicts the likelihood of finding an optimal solution on that branch, and have all our processes explore those branches until the solution is found[26]. This approach can be applied to optimistic simulation as well. The advantage here is that we *must* traverse all the “branches” (event lists) until the end of the simulation – thus all speculative work has the potential to be useful. It is only speculative in that we are uncertain when we execute it the first time whether it will need to be undone, and re-executed or cancelled. The heuristic we use here in place of “most likely to lead to a solution” in a state space search is “least likely to be rolled back” in an optimistically synchronized parallel discrete event simulation. For these reasons, we have chosen to focus our efforts on an adaptive optimistic approach to PDES.

Chapter 3

Charm++

For this thesis, we have chosen to build our PDES environment in Charm++[20] which is an object-based, message-driven parallel programming language. The basic unit of parallelism in Charm++ is a message-driven C++ object called a *chare*. A Charm++ program consists of a collection of chares that interact with each other by calling each other's methods asynchronously. Methods of a chare that can be invoked from objects on remote processors are called *entry* methods.

3.1 Parallel Discrete Event Simulation in Charm++

Charm++ supports the concept of *virtualization* very well, and we believe this approach will give rise to the great improvements in PDES performance[23]. Virtualization involves dividing a problem into a large number N of components (chares) that will execute on P processors[22]. N is independent of P, with the exception that $N >> P$. With virtualization, the user's view of a program is 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 (see Figure 3.1).

Since many chares can be mapped to a single processor, Charm++ uses *message-driven execution* to determine which chare executes at a given time. This is accomplished by having a dynamic scheduler running on each processor. The scheduler has a pool of *messages*, i.e.

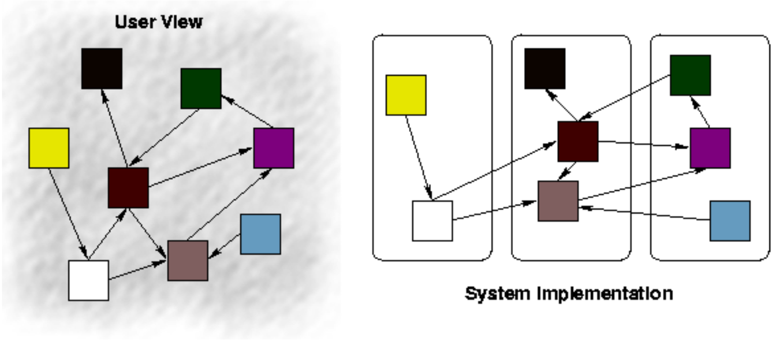


Figure 3.1: Virtualization in Charm++

entry method invocations destined for a particular chare. It selects one of these messages, determines the object it is destined for and then invokes the corresponding entry method on the object. At completion, the scheduler goes on to select the next message. Different scheduling policies are available, as well as *prioritized execution*, which enables the user to attach priorities to the 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 with work to do that can run in the interim. We discuss virtualization in the context of Pose in more detail in Chapter 5.

Object-oriented design is a straightforward means of describing most physical systems. Designing a model in Pose for a physical system is not unlike designing a model in the C++ programming language or any other object-oriented language. This commonality of design is what makes Pose accessible as a tool for PDES for multiple disciplines. We illustrate this commonality in Figure 3.2, which shows how the concepts of objects and methods on those objects map to similar concepts in Charm++ and Pose.

In a similar fashion, we can map the entities of PDES into Charm++. The logical processes (LPs) of PDES (called *posers* in Pose) can be mapped onto Charm++'s chares in a straightforward manner. Events are special timestamped entry methods on posers. We can specify the timestamps on messages as priorities and use the Charm++ scheduler as an event list. The virtualization model will assist us in providing the simulation programmer

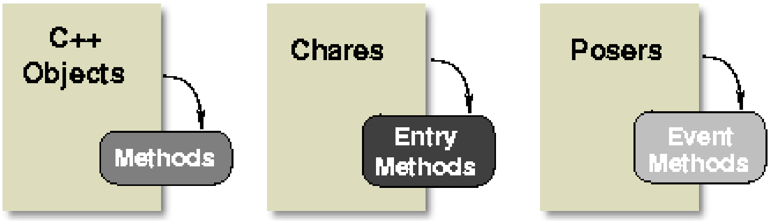


Figure 3.2: Object-based discrete event simulation

with a view of the program consisting of the entities in the model and not the underlying parallel configuration.

There are several other aspects of Charm++ that make it amenable to PDES. Charm++ provides generalized arrays of parallel objects[31, 30]. Array *elements* are objects that are scattered across multiple processors and are addressed by their *index*. Array elements can be sent messages directly via this index and they can participate in reductions and broadcasts. They can also migrate from one processor to another at any time. This capability was very useful in the design of Charm++'s load balancing strategies[5, 19]. We make use of these object migration abilities to implement our own load balancing strategies for optimistically synchronized simulations in Pose. These are discussed further in Chapter 9.

Another benefit of using Charm++ is its support for the recognition of object communication patterns[47] and corresponding communication optimization libraries. Programs exhibiting recognized patterns can be optimized to make use of common communication algorithms. In addition, changes in pattern can be handled automatically by switching to different algorithms corresponding to the new pattern[21].

For Pose, we make use of a these features to collect and periodically deliver messages by grouping them together into a single send operation. This will reduce the overhead in situations where there are a large number of very small messages being transmitted. We will discuss more strategies for communication optimizations in Chapter 8.

Charm++ also provides runtime instrumentation capabilities via Projections. Projections makes it possible to visualize performance of a parallel application in a variety of ways.

POSE additionally uses Projections' capabilities to visualize performance of the system being modelled. We discuss this in more detail in Chapter 10.

Another important reason for using CHARM++ is its highly portable nature and the existence of ports to most available parallel architectures and distributed environments.

Chapter 4

PDES Performance Analysis Methodology

Past research has demonstrated the challenges of achieving good parallel performance from fine-grained PDES. Some of the performance results are poorly-expressed and unclear. Some lack in-depth analysis of the results obtained, leaving key attributes such as problem size, granularity, number of events, base speedup comparison time, etc. unexpressed. Since it is our goal to study and improve the scalability of problematic PDES applications, we must clearly define what we mean by scalability and in what ways we will measure it.

4.1 Scalability Goals

What is scalability? We pose this question because scalability has many definitions and which one is intended depends on the background of the person using the word. Steinman[45] gives a detailed description of the various meanings of scalability in several computing contexts.

In this thesis, we are striving to achieve scalability of the *algorithm* and how it performs as a function of problem size and available computational power. Utilizing available computational power means we should be able to achieve significant speedups over the sequential simulation execution time and continue to improve the speedup as we add processors.

The key problem in this endeavor is that parallel discrete event simulation often performs poorly compared to sequential DES. In particular, the parallel synchronization of events has a high overhead which tends to dominate when the event granularity (computation time of each event) is fine. It frequently takes many processors to obtain any improvement over sequential time, if any improvement happens at all. Even if we can better the sequential time, further improvements are elusive. This fact is often finessed by reporting speedups over a one-processor parallel run. Although this is useful for isolating causes of bad parallel performance, one must also compare to actual sequential execution time (or an estimate), which incurs no more overhead than it takes to insert each event into a sorted event queue.

Minimizing parallel synchronization overhead in conjunction with fine-grained events is the most significant factor in being able to achieve larger speedups with fewer processors. While it is typically the case that in problem size scalability we wish to complete double-sized problems in double the time, this is not a sufficient requirement in PDES. Rather, we would like our mechanisms to be able to handle more work without much more overhead.

There are two key goals for our speedup desires: to achieve a lower *break-even point*, which is the least number of processors required to match or better the sequential execution time; and to obtain a greater *maximum speedup* relative to sequential execution time, where we add processors until the simulation cannot run faster and compare the best time to sequential time. To improve our break-even point, we must reduce the per-event overhead as the number of events increases. To improve maximum speedup, our PDES environment must respond to the available parallelism, optimize communication and maintain load balance.

Improving the break-even point has another purpose. While it is becoming more common to have greater computing power at our disposal, that power most likely appears in the form of a PC with 4 processors, or a cluster of 16 workstations in a lab, rather than a supercomputer with 10,000 nodes. It is disheartening then, when a sequential simulation outperforms a parallel simulation on that workstation cluster. Yet this is frequently the state of affairs in PDES.

Another approach we occasionally use is to measure *problem-size* scalability. To achieve good problem-size scalability we should be able to double the size of the problem while no more than doubling the time it takes to execute the problem. Problem size in PDES refers to the number of events executed by an application. Problem size scalability is useful to show when large problems cannot fit within the memory of just a few processors. It also reveals how well a mechanism is capable of handling more work at a lower per unit cost.

4.2 Performance Measures

This section should serve as a glossary of performance-related terms that will be used in the rest of the thesis.

Sequential Execution Time T_s is the time to execute a simulation on the POSE sequential simulator.

Estimated Sequential Execution Time T_e is an estimate of sequential execution obtained from T_i (below). If the simulation performs n events, we multiply n by a per-event overhead factor f and add it to T_i to obtain T_e. This is useful when we cannot run an application on a single processor due to memory constraints.

Ideal Sequential Execution Time T_i is an estimate of the time to execute just the events of a simulation with no intervening overhead and with an optimal ordering of the events that improves cache performance. This is a lower bound on T_s that was discovered when parallel POSE runs were found to achieve a lower grainsize per event than the sequential runs. This is most accurately measured on the fewest number of processors that the application will fit on.

T_i is our best guess at a lowest bound a simulation can have. Imagine running a simulation and logging all of the events that happen and keeping track of the dependencies between events such that we obtain a partial ordering over all the events. Now imagine reordering the events to an optimal total ordering that maintains the partial order and best reduces the number of control switches between objects, thereby obtaining the best possible cache hit ratios. Executing the events according to this total ordering would achieve a lowest possible bound on the sequential simulation time. It is impossible for a simulation to actually realize this time because not only are the total and partial ordering of the events unknown at simulation start time, but the actual events that will be executed are also unknown.

Parallel Single-Processor Execution Time T_1 is the execution time of a simulation running with parallel POSE on a single processor. It is typically larger than T_s because it includes optimistic synchronization costs associated with checkpointing, GVT calculation, fossil collection, etc.

Parallel Execution Time on k Processors $T(k)$ is the execution time of a simulation running with parallel POSE on a k processors.

Break-even Point BP is the smallest k such that $T(k) < T_s$.

Speedup $S(N)$ is the speedup on N processors defined as follows:

$$S(N) = \frac{T_s}{T(N)}$$

. We may occasionally wish to talk about *speedup relative to* other single processor time measures such as T_e, T_i or T_1 and will clearly state this when we do.

Maximum Speedup S_{max} is the maximum speedup relative to T_s obtained for an application running on parallel POSE. We frequently write "S_{max} = 17.24 on 32" to indicate that the maximum speedup we obtained for a given program was 17.24 and that this was obtained on 32 processors. This speedup may not be the absolute maximum possible, but it is what was obtained given the number of processors that were available to run the application on at the time.

Peak Efficiency Speedup S_{peak} is the largest speedup obtained while maintaining a reasonable efficiency. The choice of S_{peak} is subjective, but it is typically the point on the speedup curve after which the curve levels off or begins a slowdown. Note that runs on fewer processors may exhibit higher efficiency. Here, efficiency is defined as a percentage

$$E(N) = \frac{S(N)}{N} \times 100$$

where the speedup on N processors is divided by N.

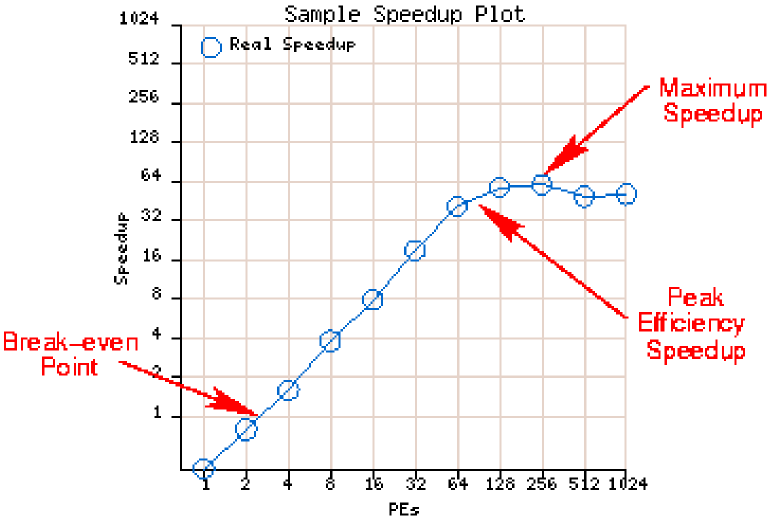


Figure 4.1: Scalability Measures

The last three definitions above are illustrated in Figure 4.1.

4.3 Factors Affecting Scalability of PDES

In this section, we discuss the factors affecting performance in PDES and then define and describe the various types of overhead that result from the use of optimistic synchronization for PDES.

4.3.1 Granularity

For every event, there is a certain amount of overhead we must endure to handle the event in parallel. If the work performed in processing the event is small, the overhead will far outweigh it. Figure 4.4 illustrates this problem clearly. Forward execution time is shown by the black section at the bottom of each bar and represents about 20% of the total execution time. We will discuss this chart further in Section 4.3.5.

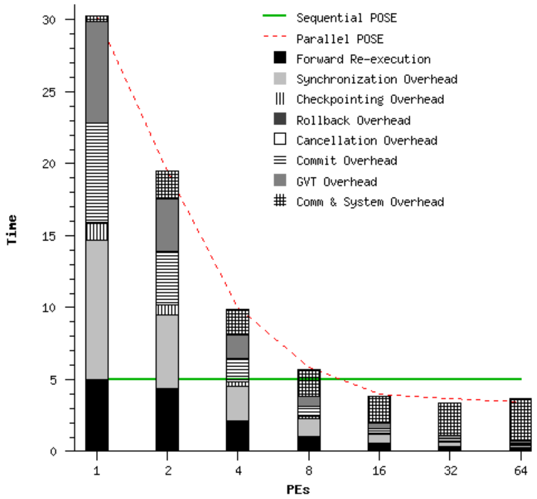


Figure 4.2: An overhead chart

4.3.2 Degree of Parallelism

The parallelism available in a simulation strongly affects the simulation's ability to scale. We discuss the degree of parallelism in the context of a synchronization strategy with a *time window* that limits how far ahead of the GVT events can be executed.

If we were to create a set of timelines for each object, view the position of each object on its line as the simulation progressed and connect all the objects that have unprocessed events to execute within the time window, we'd get what we call the *execution front* for the simulation. It would be a slightly wavy line near the GVT estimate t. Examining this line and the events that succeed it allows us to measure the extent and scope of parallelism at this point in the program. The *degree of parallelism*, $D(t)$, is the number of objects on this line. Figure 4.3 illustrates the execution front.

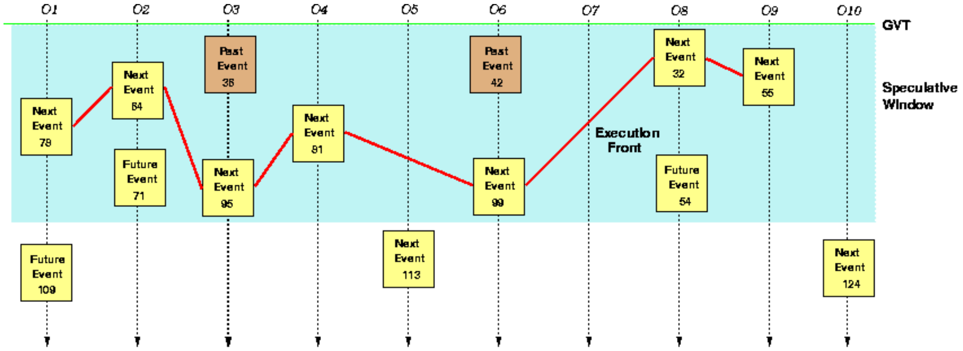


Figure 4.3: An execution front in a simulation with 10 objects; here $D(GVT) = 9$ and $D(GVT, w) = 7$

Since we are limiting forward execution by allowing it to happen only within the bounds of a time window, we further parameterize the degree of parallelism with the window size w. This results in $D(GVT, w) = 7$ in Figure 4.3. $D(GVT, w)$ represents the potential for parallelism at a time t but does not take into account how the objects are distributed across the processors of a parallel machine.

We wish to calculate the degree of parallelism on P processors for a particular window size w, or $D_P(t, w)$. A simple metric for this is to simply count the number of processors with at least one object with work within the time window. If $P = 2$ and we partition the objects in Figure 4.3 down the middle between $O4$ and $O5$, both processors has some work to do, so $D_2(t, w) = 2$.

However, we would like $D_P(t, w)$ to be a more meaningful representation of how well the available parallelism has been realized. Given the global degree of parallelism $D(t, w)$, we determine an average load per processor $A_avg(w, t)$ that would keep all processors busy working in parallel (assuming all events are of a similar grainsize and that there are no additional events within the window beyond the execution front) for as long as possible. Thus,

$$A_avg(w, t) = \frac{D(t, w)}{P}$$

We can now calculate $D(t, w, i)$ as the degree of parallelism on processor i, out of the maximum $A_avg(w, t)$. Let $A_i(w, t)$ be the number of objects on i at time t with work to do within the time window. Then

$$D(t, w, i) = min\{A_i(w, t), A_avg(w, t)\}$$

We can now obtain

$$D_P(t, w) = \sum_{i=1}^{P} D(t, w, i)$$

From this we can calculate a percentage of the available parallelism that was realized by a particular distribution of objects as $D_P(t, w)/D(t, w)$.

Trying to improve degree of parallelism measures is a major goal of this research. For this, $D(t, w)$, should be maximized. This is largely in the application developer's hands. Large, active objects should be decomposed into smaller objects that can execute events in parallel. Another way of increasing $D(t, w)$ is to adaptively expand the time window size w thereby making more work available for execution. This induces a trade-off because the larger window increases the likelihood of receiving a straggler event. We introduce speculative and adaptive strategies in Chapter 6 that attempt to maximize $D(t, w)$ while managing the trade-off as best as possible.

$D_P(t, w)$ should be as close to $D(t, w)$ as possible. To achieve this, we make $D(t, w, i)$ for each processor i as close to average as possible. This means we must have a good mix (in terms of timestamp ranges) of available work on all processors. If all processors have roughly the same amount of work within the time window as the window moves forward through time, then they will all approach average $D(t, w, i)$. When this work is evenly distributed over time, we can achieve a longer period of parallelism between GVT calculations, which happen when a designated processor runs out of work to do within the window (more on GVT in Chapter 7). Low $D_P(t, w)$ could indicate a situation where some processors have less work or none, causing them to go idle and bring down the average processor utilization. While programs with a low degree of parallelism throughout will always be problematic, we still hope to find ways of improving the performance in these cases. We use the load balancing techniques described in Chapter 9 to maintain an average $D(t, w, i)$ on all processors.

4.3.3 Object Size

We mentioned in the previous chapter that the global state is partitioned among many LPs that are objects known as *posers* in POSE. The *size* of a poser is a combination of the size and complexity of its data and the quantity and size of event messages it receives. Large objects take more time and space to checkpoint, more time to perform fossil collection on and more time to migrate. More events means more overhead to handle them and more space to store the data associated with them. In addition, decomposing a large object into smaller objects could improve the degree of parallelism.

The fewer objects that exist, the more important it is that each one is very active. This is another reason why having $N >> P$ (from our discussion of virtualization in Chapter 3) is so important. If we can decompose the simulation so that it has more active objects, we can improve the degree of parallelism.

There are several other factors that affect performance in PDES. Everything from object decomposition and distribution across processors to patterns of communication can have an impact. In the next section we discuss the activities performed in optimistic PDES and how the nature of the simulation model affects them.

4.3.4 Sources of Overhead

What must we do in the parallel DES case that isn't necessary in the sequential case? In Bagrodia's 1996 paper [3], three primary sources of overhead in PDES are listed: partitioning-related, target architecture, and synchronization protocol overheads. We will address the first two in the sections on communication and load balancing. Here our focus will be on the overhead of the optimistic synchronization protocol. We have categorized the various types of overhead encountered in PDES and illustrate their significance in the overhead chart for a Pose program shown in Figure 4.4.

Forward Re-execution Overhead: *Forward execution* is the process of executing the next minimum timestamp event in an event queue. This is the actual work of the simulation, so we do not regard this as overhead. However, a portion of the forward execution time is spent *re-executing* events that were previously rolled back. We can reduce this overhead by decreasing the amount of speculative computation an object is allowed to perform. Since this is directly related to rollback overhead, we will discuss this in more detail under that heading.

Message Sending Overhead: Another source of overhead in the forward execution phase is in the issuing of remote events. Discounting the actual time to transmit the event, we must still package the data to be sent in the event message, apply a timestamp to that message, and possibly perform other operations. For example, we may be recording information about each event generated to facilitate the operation of the GVT algorithm, or the load balancer. In a forward execution step for a very fine-grained operation, simply issuing another event may be the bulk of the computation performed.

Checkpointing Overhead: Checkpointing overhead is the time it takes to checkpoint an object's state before an event is executed which might change that state. Various strategies exist for when and how to checkpoint. In one approach, the *entire* object gets checkpointed before *every* event. We call this *full checkpointing*.

This may seem like a wasteful approach, but let's examine the factors contributing to this overhead. The most obvious is the size of the object, in terms of the number of bytes of data that must be checkpointed and the complexity of the data structures that must be traversed to do so. Clearly, when the data is large and complex, the full checkpointing approach can be both time– and space–consuming. However, to mesh with the virtualization concept, we should decompose large data objects into smaller objects thereby increasing the degree of parallelism, while simultaneously reducing the checkpointing overhead.

Some further methods to reduce this overhead are *partial checkpointing* and *periodic checkpointing*. In partial checkpointing, we checkpoint only a portion of the state that may have changed. Drawbacks to this are the overhead of determining during reconstruction what changed during forward execution and how much of that burden is placed on the user. In periodic checkpointing, we only checkpoint before some events. We may checkpoint once per K events, or use some other criteria to determine when we should checkpoint the state. This creates other potential sources of overhead, such as having to reconstruct a state halfway between two checkpoints. It also makes GVT estimation more challenging.

Synchronization Overhead: Synchronization overhead could be thought of as anything not covered by the other categories. This is typically the time it takes to orchestrate all of the other activities that occur. It includes the time spent receiving events on a poser, inserting events in the object's event queue, determining when and what events can be executed in the synchronization strategy, determining if a rollback is necessary or if a cancellation is pending, etc. This type of overhead is also present in sequential simulation, since events need to be sorted by timestamp.

Synchronization 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 synchronization overhead will be large relative to the forward execution time.

Rollback Overhead: Rollback overhead occurs when a straggler event arrives and the simulation must undo events, send cancellation messages to other spawned events and restore the checkpointed state prior to the straggler. As the number of processors rises, so does the chance that sequencing errors will occur. The use of speculative synchronization strategies also increase the likelihood of rollback.

Cancellation Overhead: Cancellation overhead includes the time taken to receive and handle event cancellation messages. Rollbacks caused by cancelled events are included in rollback overhead. Cancellations generated by rollbacks can be reduced through the use of *lazy cancellation* which only cancel events if they are not regenerated in exactly the same way.

Commit Overhead: Commit overhead is mostly fossil collection time. We perform fossil collection whenever a new GVT estimate is available. Here is another advantage to full checkpointing — we can commit all events with timestamp less than the GVT estimate. If checkpointing were periodic, we'd only be able to discard events up to some checkpointed event with a timestamp less than the GVT estimate. Depending on how frequently the checkpoints occur, this could mean a lot of memory remains in use per object, or that a lot of time is spent searching for the appropriate point in history to commit up to. As in the case of checkpointing, the size of the object is also a factor in determining the commit overhead.

GVT Overhead: GVT overhead is the amount of time spent gathering and organizing data and using that data to compute the GVT estimate. A great deal of information is required from all objects in the simulation to accurately determine a single estimate. Thus, 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[32, 33, 46, 35].

GVT computations are especially dominating in computations with a low degree of parallelism. In Pose, speculative computation is linked to good GVT estimates, but a low degree of parallelism means even speculative work is hard to come by. Thus the GVT computation runs constantly, trying to obtain better estimates to enable more work that isn't there.

Load Balancing Overhead: When we make use of the Pose load balancer, we have another source of overhead that we hope will be negligible in comparison to the improved processor utilization. Load balancing overhead is dependent on the balancing strategy being used — the number of times load balancing is invoked, the number of object migrations that take place, etc. Again, size of an object is a factor in how long it takes to pack up, transmit and then unpack a single object.

Communication and System Overhead: This overhead is any time we cannot account for inside of the POSE code. It includes time for message packing/unpacking and prioritized scheduling in CHARM++, idle time spent waiting on a processor for work to do, any operating system interference, etc. This overhead tends to have an erratic behavior as the number of processors change, but we have seen it exhibit a trough-like pattern: high on few processors, high on many, and low in between. The communication component of this overhead naturally increases with the number of processors, since the percentage of non-local communication must rise. On the other hand, the system scheduling overhead scales down as the number of processors increases because per processor queue lengths decrease.

4.3.5 Synthetic Benchmark

Examining the overhead of synchronizing events in parallel gives us the opportunity to identify, study and classify challenging problems for performance improvement in PDES. Such study is applicable to all aspects of performance of PDES, from improving speedups and scaling to larger numbers of processors, to improving processor utilization and reaching the break-even point on fewer processors.

To better illuminate these problems, we designed a synthetic benchmark parameterized to exhibit the wide variety of behaviors found in PDES simulations. This benchmark takes the parameters specified in Table 4.1.

The benchmark creates n objects, places them on processors according to an initial distribution d and each object then sends a work event to itself to start things off. The work event consists of performing some computation for the time specified by the granularity g, then elapsing some time according to the communication pattern p, and finally spawning up to e work events for other objects with message size s, using connectivity c, locality l, and communication pattern p to determine where and when those events take place. The simulation proceeds until we reach t in virtual time.

Number of Objects n
Maximum number of events e spawned per unit of work done
Message size s (small, medium, large or mixed)
Initial distribution of objects d (uniform, random or imbalanced)
Communication connectivity of objects c (sparse, heavy, full)
Locality of communication l (percentage local communication)
End time t (time units to run the simulation)
Granularity g (fine, medium, coarse, mixed, or a constant)
Communication pattern p(controls how time is elapsed or offset, and how an object spawns events)
Object data size (affects checkpointing and migration)

Table 4.1: Parameters to Synthetic Benchmark

The behavior of this benchmark can change dramatically over time. The communication pattern here is key. Note the the number of events generated in each work event is "up to e". The communication pattern controls the density of communication. At its densest, the number of events in the benchmark will keep increasing dramatically. The pattern can also allow the total number of live events to dwindle or remain in a steady state. It further controls how events are dispersed throughout time: they may come in waves, or they may be dispersed randomly throughout time.

We ran this benchmark with an early version of Pose using a parameter set with high parallelism but fine granularity[1]. Figure 4.4 illustrates the overhead for this problem.

This chart shows the average time per processor for each type of overhead in a simulation. Here we see that the application breaks even with sequential time just past 8 processors, then the performance levels out quickly between 32 and 64 processors. Forward execution is approximately $\frac{1}{6}^{th}$ of the time for all runs while the rest is overhead.

We will use this type of chart to examine overhead in simulations frequently in this thesis.

[1] Our synthetic benchmark was run on Lemieux (see Section 4.4) with: 12000 objects, a maximum of 5 events spawned per unit of work done, SMALL message size, UNIFORM distribution of objects, SPARSE connectivity of objects, 50% locality of communication, run for 500 time units, SMALL grainsize (0.000001s of work performed), and a highly irregular communication pattern. The measured granularity was approximately 0.0004s on average.

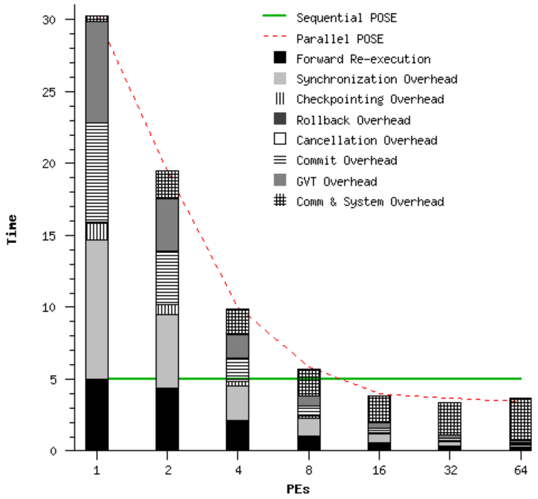


Figure 4.4: An overhead chart

4.4 Parallel Computing Environments

The experiments in this thesis were conducted in the following environments. For a given set of results, we will use the one word descriptor shown here in boldface.

1. **Lemieux** is a machine at Pittsburgh Supercomputing Center (PSC) consisting of 750 Compaq Alphaserver ES45 computational nodes each containing four 1-GHz processors running the Tru64 Unix operating system. A Quadrics interconnection network connects the nodes. Each node is a 4 processor SMP, with 4 Gbytes of memory.

2. **Turing** is a Center for Simulation of Advanced Rockets (CSAR) cluster of 640 Apple Xserves each with two 2 GHz G5 processors and 4 GB of RAM connected by Myrinet.

3. **Tungsten** is a National Center for Supercomputing Applications (NCSA) Linux cluster of 1280 nodes of dual Intel Xeon 3.2 GHz processors connected by gigabit Ethernet and Myrinet 2000.

4. **Cool** is a Linux cluster of eight nodes of quad Xeon SMP connected by fast Ethernet and maintained by the Parallel Programming Laboratory at the University of Illinois at Urbana-Champaign.

5. **Arch** is a Linux cluster of 75 nodes of dual Athlon MP 2000+ processors connected by gigabit Ethernet and maintained by the Computer Science Department at the University of Illinois at Urbana-Champaign.

Chapter 5

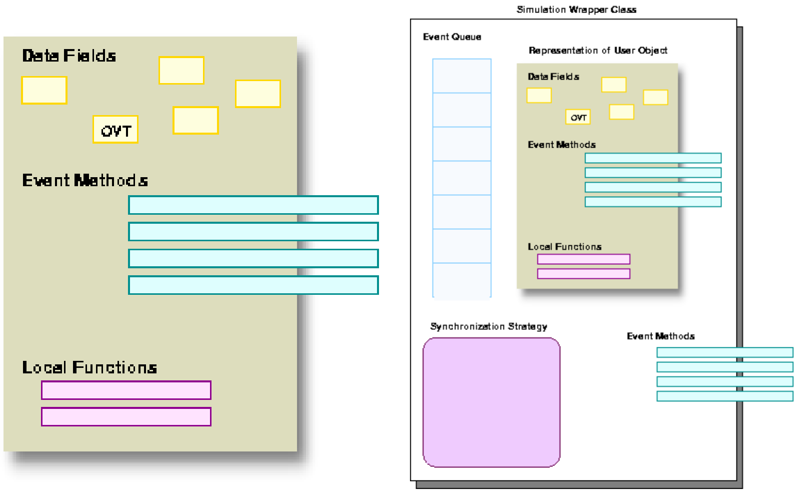
Localizing Overhead Through Virtualization and the POSE Object Model

In this chapter, we discuss the concept of virtualization and how we make use of it in POSE. This leads us to the development of the underlying design of POSE and how it will benefit us in the simulation of challenging fine-grained problems.

5.1 POSE Objects

As mentioned earlier, we have built our PDES environment in CHARM++ as it supports the concept of *virtualization*. In POSE, simulation entities in a model are called *posers*. Posers are special chares that represent portions of the global state of a simulation model. From the user's perspective, the differences between posers and chares are few. The first difference is that each poser has a data field for *object virtual time* (OVT). This is the number of simulated time units that have passed since the start of the simulation relative to the object.

Another difference is that posers have *event methods*. Event methods are nearly identical to the CHARM++ *entry methods* discussed earlier, with the main difference being the presence of a data field for *timestamp* in all messages sent to invoke an event method. Figure 5.1(a) shows the user's view of a poser and its possible method types.



(a) (b)

Figure 5.1: (a) User's view of a poser; (b) Internal Pose representation of a poser

Posers thus represent the LPs of traditional PDES: they encapsulate a portion of the global state and all the processes that can operate on that portion. They also have a local clock in the form of the OVT. We will also see shortly how each poser also has its own event queue.

Posers have two ways of passing virtual time. The first method is via the *elapse* function which is called by the user code. This is used in a poser when a certain amount of local virtual 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. It determines the timestamp on the event that is sent. For example, suppose the event being invoked involves the movement of data (such as a packet being sent over a network) and it takes t time units to transmit that packet, we would schedule an event at the point receiving the packet at a time that is offset by t from the current time. This could cause a the receiving poser to elapse time. Suppose the receiving poser has an OVT of 3492 and the received event is timestamped 4287. If that event is the next in the poser's event queue, it will assume that nothing else is going to happen between 3492 and 4287 and the poser's OVT will advance to 4287 and it will execute the event.

The internal representation of a poser consists of the user's representation encapsulated in a special control object. This object combines the state represented by the user's object with an event queue and an instance of a synchronization strategy. The control object has entry methods which correspond to all the events that the user's object can receive. These entry methods capture incoming events, store them in the local event queue and invoke the local synchronization strategy on them. The event queue is a multipurpose data structure that also stores checkpoints for the user's object's state. Figure 5.1(b) illustrates the internal POSE representation of a poser.

Why encapsulate so much information in each entity, and then encourage the breakdown of the physical system into many such entities? Using virtualization allows us to maximize the degree of parallelism. By giving each poser the functionality of an LP, we have completely virtualized the driving mechanisms of PDES for better performance. Including an event queue in the object itself means that the scope of all 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 much larger, more static entity, we would want to separate it from the larger construct and thereby avoid checkpointing a much larger state. Further, encapsulating all the relevant data for an object within it makes migration of that object much simpler.

5.2 Events

An event is an invocation of an event method via an event message. In addition to a timestamp, the structure of a POSE `eventMsg` contains several other data items. Each event has an integer *priority* that is used by the CHARM++ scheduler to determine the order in which incoming messages will be handled. We use the timestamp of the event to determine what the priority will be, giving earlier timestamps higher priority.

An event also has a unique system-wide *event identifier*, the `eventID`. This is used to distinguish events when they must be relocated or cancelled and to provide determinism to the ordering of events with the same timestamp. It also has a field indicating which processor sent the event which is used by the GVT algorithm and another field for recording the size of the message which is used in message recycling.

When an event message is received on the object, it is stored in an event queue where it stays until the event is committed. All data needed to undo the event, such as the checkpointed state of the object, a list of spawned events, etc. is stored in the event queue entry for the associated event.

5.3 Synchronization

Some posers in a simulation may be more suited to a particular synchronization behavior. For example, one type of entity may simply change state periodically regardless of outside influence and generate events to notify other entities of these changes. Such an entity might benefit from a more restrictive synchronization behavior that keeps it in step with the advancement of the GVT. Thus, it would be useful for each entity in a model to have its own instance of a synchronization strategy which is tuned to the entity's behavior. Further, as the behavior of an entity evolves in a simulation, the strategy could adapt its handling of events to be better suited to the new behavior of the object. We discuss our approach to synchronization in detail in the next chapter.

5.4 POSE Simulation Structure and Process

As we have seen, most of the PDES processes that happen are contained within the POSE object model. We now describe how the simulation process works and what other components exist in POSE.

POSE provides a source-to-source translator that takes the user's class and translates it to the internal representation. The encapsulating class that is created derives from the base class `sim`. For example, if the user creates a poser called *MyEntity*, then a class of that name is created that derives from `sim`. Then, another class *state_MyEntity* is created which derives from the `rep` base class and is encapsulated in *MyEntity*. The encapsulated class *state_MyEntity* holds all the data fields and provides all the methods of the user's original specification of the poser. The translator duplicates the event method interface of the original poser for *MyEntity*. *MyEntity* also contains an event queue and a synchronization strategy.

The startup of a POSE simulation involves an initialization phase. The GVT is started and runs asynchronously with the simulation. A CHARM++ chare array is created which will contain all the posers in the simulation. Posers are thus addressed via an array index. Any other included components, such as communication optimizations, load balancing, statistics collection, etc. are initialized. The main program creates the posers. Startup events can be initiated in the main program, or from poser constructors.

Events are generated by calling event methods on a poser. These are passed an `eventMsg` or a message that is derived from the `eventMsg` type. This message has both a timestamp and a priority based on its timestamp. When an eventMsg arrives on its destination processor, it is enqueued according to its priority in the CHARM++ scheduler. The scheduler processes these events one at a time. It selects the message with highest priority and calls the event method corresponding to the message. This would be one of the duplicated event methods generated for the encapsulating class (*MyEntity* above). All of these duplicate event methods perform the same function: they enqueue the event on the poser's event queue and invoke the synchronization strategy.

The synchronization strategy then checks the state of the poser for several conditions: 1) have any cancellation messages arrived (there is a special queue just for these on the poser); 2) have any straggler events arrived. These situations may result in the removal of events from the poser's event queue and/or a rollback of events. Once handled, the strategy then attempts to perform forward execution. It examines the earliest event in the queue and executes it according to the strategy's criteria (detailed in the next chapter). Execution may involve several steps before and after the actual execution of the user's event method on the *state_MyEntity* object. For example, the state of the object may need to be checkpointed and various statistics collected about the object's behavior that will be used by the strategy, the GVT algorithm and other components of the simulation. The strategy may choose to execute several events or it may also choose not to execute any events depending on the state of the poser.

For checkpointing, the user has several behaviors to choose from. POSE implements full checkpointing, periodic checkpointing and a new approach via *anti-methods*, which allow the user to specify functions to undo simple state changes.

Rollbacks involve determining the point to roll back to, undoing events by issuing cancellations for spawned events and restoring state either from checkpoints or by calling an anti-method for each undone event. In the case of periodic checkpointing, the point to which we wish to rollback may occur between checkpoints. POSE uses a lazy approach to recover the state that does not undo the events between the rollback point and the checkpointed state. Cancellation may involve rollback, but if the cancelled event was not executed, the event can simply be deleted.

All of these operations happen at the poser level and are thus operating on data sets that are much smaller than if we applied them at the processor level.

5.5 Effects of virtualization

As mentioned earlier, posers are meant to encapsulate smaller portions of the global state than traditional LPs. POSE's object-oriented nature enforces this structure by requiring that every distinct sequential entity in a simulation model be a poser, but the responsibility for decomposing a simulation model into the smallest components possible (and thereby the largest number of posers) lies with the programmer implementing the model in Pose.

In addressing the issue of fine-grained computation, this approach may seem counterintuitive to a more natural clustering together of entities. However, decomposing a simulation model into many components with the poser structure has many benefits. Including an event queue in the poser 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. When checkpointing is performed periodically according to the number of events received by a poser, this structure enables less frequent checkpointing on smaller states. This reduces overhead as well as memory usage. Rollbacks are less likely to occur when the scope effected by incoming events is much smaller. Further, encapsulating the relevant data in smaller lightweight objects makes migration of LPs much simpler. Since each poser has its own instance of a synchronization strategy, this structure paves the way for adaptive synchronization strategies that are finely-tuned to the behaviors of each small LP.

The potential drawbacks of a high degree of virtualization are the management of information for a much larger quantity of simulation entities, coupled with the cost of switching from one such entity to another for each event.

Thus, the performance benefits of virtualization alone are significant, but we need to examine what tradeoffs might exist. We ran a series of experiments in which we varied the degree of virtualization of a fixed-size problem using only a very basic optimistic synchronization strategy with no throttling mechanisms in place. We wanted to see how virtualization could improve performance by reducing the overheads previously mentioned, but we also wanted to measure the increased overheads at the highest degree of virtualization. These experiments were executed on the Cool cluster with a synthetic benchmark that allows parallel

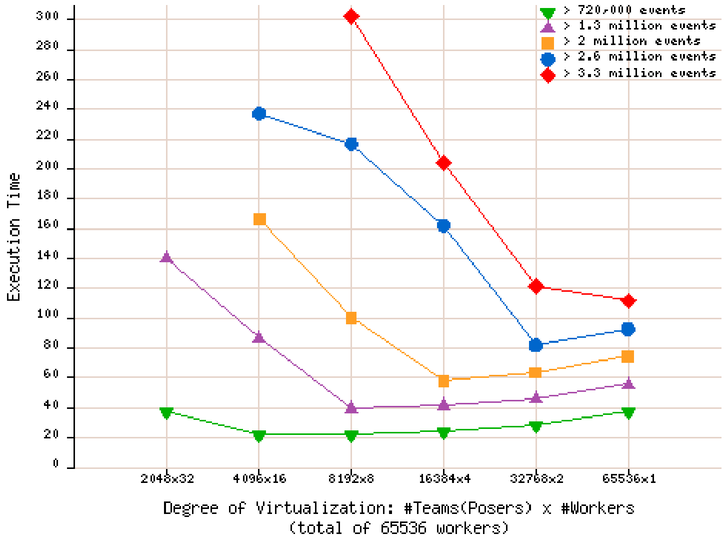


Figure 5.2: Effects of virtualization on execution time for several problem sizes on 16 processors.

workers to be grouped together into `team` posers (*i.e.* each team corresponds to a logical process). Each worker has a single event type. When it receives an event, it performs some busy work, and generates an event for another worker. The event communication pattern has each worker a part of a ring of workers, such that 50% of the workers communicate locally and the rest communicate with a worker on another processor. We simulated a total of 65,536 workers in each experiment, but grouped together in different sized teams. We started with a low degree of virtualization, 2048 teams of 32 workers each, and ended with maximum virtualization, 65,536 teams of 1 worker each. We ran these experiments on 16 processors on 4 nodes of the Cool cluster with several different program sizes (in terms of number of events handled). The results are shown in Figure 5.2.

We found that the benefits of higher degrees of virtualization strongly outweighed the added costs. Further, as the problem size increased, higher degrees of virtualization consistently outperformed lower degrees of virtualization. It should be noted that we designed our benchmark to make rollbacks very unlikely, so what we see in Figure 5.2 is purely the cost of optimistic synchronization running optimally. Less optimal simulations with rollbacks would be more likely to incur higher costs for lower degrees of virtualization. Note that missing points for a curve indicate that those degrees of virtualization either ran out of memory or ran out of the time allotted for the run.

	Teams × Workers per Team				
Cost Type	4096×16	8192×8	16384×4	32768×2	65536×1
Forward Execution	38.18	36.19	31.15	31.64	32.33
GVT	61.22	11.26	1.09	1.11	1.27
Synchronization	1.35	2.37	1.08	1.24	1.42
Checkpointing	3.96	3.45	0.98	0.78	0.55
Fossil Collection	15.40	12.70	1.78	1.89	2.00
Rollback/Cancellation	0.00	0.00	0.00	0.00	0.00
Communication	14.57	14.00	13.98	14.42	15.62
Other	16.28	7.90	6.10	10.90	19.20

Table 5.1: Breakdown of costs in seconds (averaged per processor) for a problem size of >2 million events with varying degree of virtualization.

We show a breakdown of the performance in Table 5.1. The optimal performance for this set of runs occurs when we have 16,384 teams of 4 workers each. Activities which are highly dependent on the quantity of objects in the simulation (such as GVT, synchronization and fossil collection) are slightly elevated as the degree of virtualization increases from there. The most significant type of overhead affected by higher degrees of virtualization is the "Other" category which includes the cost of managing the objects in the Charm++ run-time system. Checkpointing costs decrease consistently as degree of virtualization increases. The most significant performance differences are attributable to the significantly higher memory usage for lower degrees of virtualization. In this experiment, as memory becomes scarce, the types of overhead that frequently deallocate and, to a lesser extent, allocate memory are most affected. These include GVT, fossil collection and forward execution. A small amount of rollback also occurs in the lowest degree of virtualization runs. It is not enough to result in significant rollback overhead, but there is some event re-execution cost included in the forward execution times.

What Figure 5.2 does not show is that the higher degree of virtualization also allows us to run the program on more processors than we can with the lower degree of virtualization, further increasing the chances of obtaining a better speedup.

Thus, this “fine granularity of entities” does not have a significant additional cost to it and has further benefits that we are not yet taking advantage of. For example, having more fine-grained entities should provide us with a great benefit when we wish to use load balancing in our simulations. As we shall see in the next section, high degrees of virtualization will enable fine-tuned adaptive synchronization.

The reasons for this effect are straightforward. In common parallel computing applications (*e.g.* 5-point stencil), if you combine objects, you reduce the number of messages between them. However, in PDES even internal messages have to go through a sorting process along with incoming external messages. We are primarily concerned with entities which can operate in parallel with other entities, no matter how small they may be. Grouping potentially parallel entities together means binding their behaviors together. The single entity must handle more events than each of the separate entities and the effects of those events apply to a larger state than those on the separate entities. In addition, the object’s queues for storing events, past and present, and cancellations, are larger as well, resulting in higher costs for all operations on them. Thus, even though the simulation is the same – the same total quantity of events must be processed – the partitioning into smaller entities reduces the impact of each of those events.

5.6 Scalability of overhead

Figure 5.3 shows how each overhead component of optimistic synchronization scales with increasing numbers of processors. This graph shows the speedup of a component relative the the time spent on that component on one processor. Each processor spends some amount of time on the overhead component in question. These times are averaged over all processors and compared to the time taken on one processor. For example, our program spends 150s doing GVT-related computation on one processor. When running on two processors, we discover that 79s was spent on PE 0 and 71s were spent on PE1 for GVT calculations. Taking the average, we get 75s which gives us a speedup of 2 on 2 processors for GVT overhead. Missing from the graph are rollback and cancellation time, which were negligible for this run and exhibit a fluctuating behavior (independent of the number of processors on which the simulation was executed) rather than a scaling behavior. Also missing is the "Other" overhead category (representing idle time, communication overhead, operating system time) which was significant for this run, but also fluctuates and does not exhibit a scaling behavior (i.e. the time does not reduce as we add processors; in fact communication time usually increases).

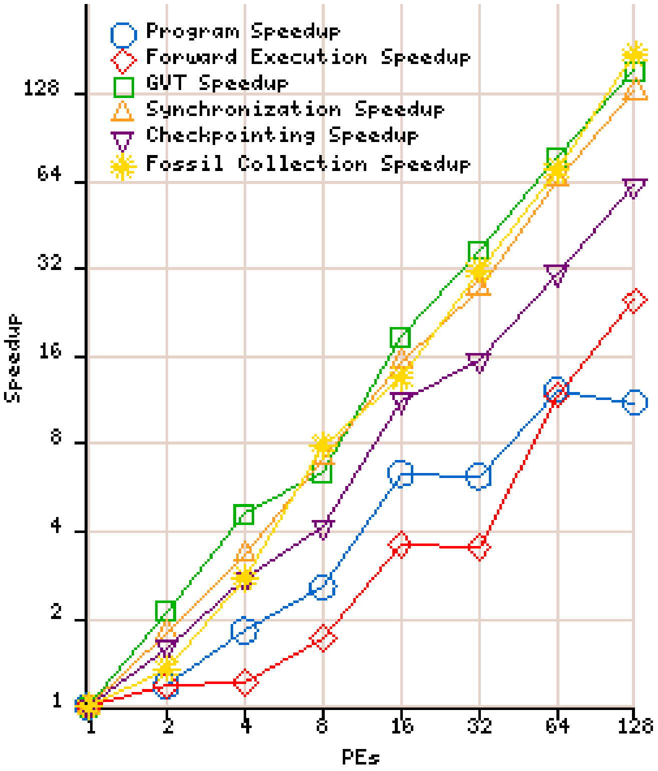


Figure 5.3: Scaling of overhead in optimistic synchronization

The synthetic benchmark program instance used for the graph of Figure 5.3 was executed on Tungsten using our most basic optimistic synchronization strategy. We used 30,000 objects and over 3 million events were exchanged. The average grainsize of an event was 4 μs. The graph shows that most activities scaled very well, particularly GVT, synchronization and fossil collection. Checkpointing had problems on fewer processors but as the number increased, the speedup improved. None of these betray the poor speedup of the entire application, except possibly the forward execution time. It is less scalable because it includes forward re-execution time which is likely to increase as we add processors. It is also affected by the parallelism in the application and any load imbalance that may exist. In addition, the average grainsize of events increases as we add processors, because of the increased time to invoke remote events. However, the largest overhead is the "Other" overhead not shown in the chart. Load balancing, communication optimizations, and simply having enough work to do as we increase the number of processors are the best approaches to tackling this problem. Table 5.2 shows the actual values of the overhead for this problem, including the missing overhead components.

	Processors							
Component	1	2	4	8	16	32	64	128
Total	65.28	55.57	35.86	25.38	10.33	10.56	5.32	5.88
Forward Execution	13.92	11.77	11.53	8.12	3.85	3.96	1.18	0.56
GVT	39.69	18.74	8.66	6.21	2.13	1.07	0.51	0.26
Fossil Collection	6.98	5.16	2.53	0.88	0.52	0.22	0.10	0.04
Synchronization	5.21	2.92	1.56	0.73	0.34	0.19	0.08	0.04
Checkpointing	1.24	0.78	0.45	0.30	0.11	0.08	0.04	0.02
Rollback	0.00	0.02	0.06	0.02	0.01	0.07	0.01	0.00
Cancellation	0.00	0.01	0.01	0.00	0.00	0.00	0.00	0.00
Other	0.00	15.62	10.31	1.42	3.09	0.92	1.77	3.56

Table 5.2: Average time in seconds for overhead components

5.7 Summary

We have shown that the use of virtualization and the design of the object model in Pose to exploit virtualization have resulted in many benefits for PDES performance. Increasing the degree of virtualization makes it possible to achieve better parallel performance, and in particular, better break-even points. Localizing the effects of events to smaller objects and thereby smaller portions of state, reduces the per event overhead and makes the overhead itself easy to scale. This object model also affords us the opportunity to develop flexible synchronization strategies and finely-tuned load balancing strategies as we shall see in the remainder of this thesis.

Chapter 6

Speculative and Adaptive Synchronization

In this chapter, we discuss our new approaches to optimistic synchronization and how they help to improve PDES performance. We developed several strategies that were precursors to our final adaptive strategy, including a simple conservative approach, a basic optimistic mechanism and several optimistic variants. All the strategies are implemented within the virtualized object model discussed in the previous chapter and benefit from the effects described there. The strategies we discuss are basic *Optimistic*, *Batched Optimistic*, *Throttled Optimistic*, *Speculative*, basic *Adaptive* and finally, the *Adept* strategy. We discuss these approaches and their performance in the remainder of this chapter.

6.1 Basic Optimistic Strategies

In Pose, an object gets control of a processor when it either receives an event or cancellation message via the Charm++ scheduler, or when the GVT component awakens the object after a new GVT estimate has been calculated. 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. Figure 6.1 shows how timestamp-prioritized event and cancellation messages are redirected to the intended objects on a processor. An event message is moved to the local event queue of the intended object where it is either handled or stored for later execution. Cancellations are moved to the cancellation list of the intended object. They will be examined before the object next performs any forward execution. The figure also shows a GVT-related message which has a priority lower than all events and cancellations. It is handled by the local PVT or GVT objects when the processor is otherwise idle. GVT estimation will be discussed in the next chapter.

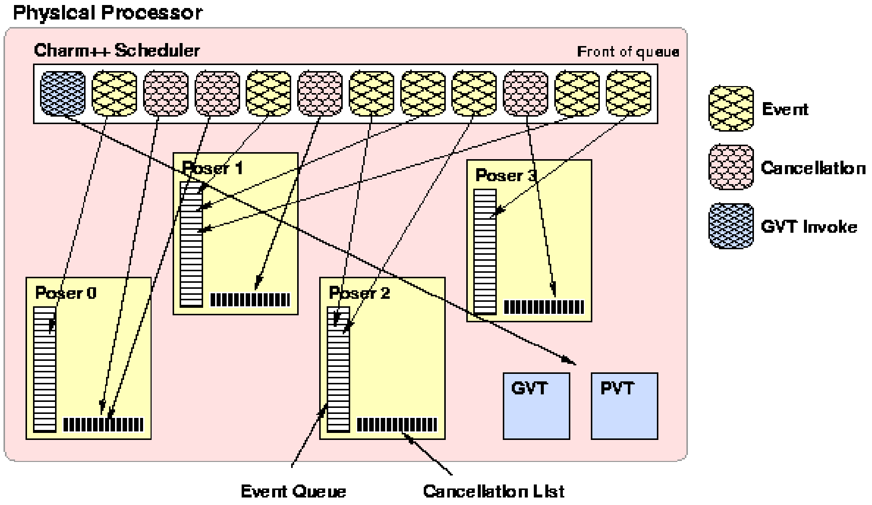


Figure 6.1: Processing messages in Pose

In the sections that follow, we discuss what happens to events after they are place in a poser's event queue. This is where the distinctions between Pose's synchronization strategies occur.

6.1.1 The Basic Optimistic Strategy

Our basic Optimistic synchronization strategy first checks for any cancellation messages it may have received and handles those whose corresponding events have arrived. When cancellation messages arrive, they are stored on the object in a list. Some may have arrived before their corresponding event; thus we need to check the cancellations to see if any correspond to newly received events before the new events are processed. Next, the strategy checks for any stragglers that may have arrived and if so, rolls back to the earliest. Finally, it is ready to perform a forward execution step.

The poser's synchronization strategy takes the earliest event in the poser's event queue and executes it. When completed, the strategy examines the timestamp of the next earliest event in the event queue and (if the queue is not empty) sends a prioritized *token message* to itself to schedule the next event for execution. The priority of the token message corresponds to the timestamp of the event in the object's queue. Thus, Optimistic has this third means by which objects get control. The token messages are prioritized in the CHARM++ scheduler and when one is handled, it again invokes Optimistic on the object to process the next event. In this way, Optimistic always processes events in best-first order on a processor (not just on a poser) given what is known on that processor at any particular time.

We show performance results in the form of speedup graphs for each of our strategies for a small and large synthetic benchmark. The speedup is relative to sequential execution time (T_s) which in the case of the small benchmark is $78.8s$ and $2824.8s$ for the large benchmark.

Small benchmarks were run with 30000 objects with a modest density of events (approximately 1600 events per virtual time unit) and a total of over 3.2 million events. Large benchmarks had 50000 objects, a high density of events (approximately 8500 events per virtual time unit) and a total of over 17 million events. All experiments were run on Tungsten (see Section 4.4) up to as many processors as it took for the speedup curve to level off or drop. This typically occurred when the available work (forward execution time) per processor dropped below 2-3 seconds.

We used the performance results for Optimistic to determine what directions we should take in designing new synchronization strategies. Figure 6.2 shows how Optimistic scales for a small instance of the synthetic benchmark relative to sequential time, as well as for a larger instance. While the smaller instance performs unimpressively, it does at least break even right away. For the larger instance, we achieve more impressive results. The curve is roughly linear until the point where there is very little work remaining per processor ($2.74s$ on average per processor on 128 processors). However, Optimistic uses more memory (due to checkpoints) than the sequential simulator so we were unable to obtain results for the large benchmark below 8 processors. We will see this problem in greater detail when we compare Optimistic to Adept using a different benchmark later in this chapter.

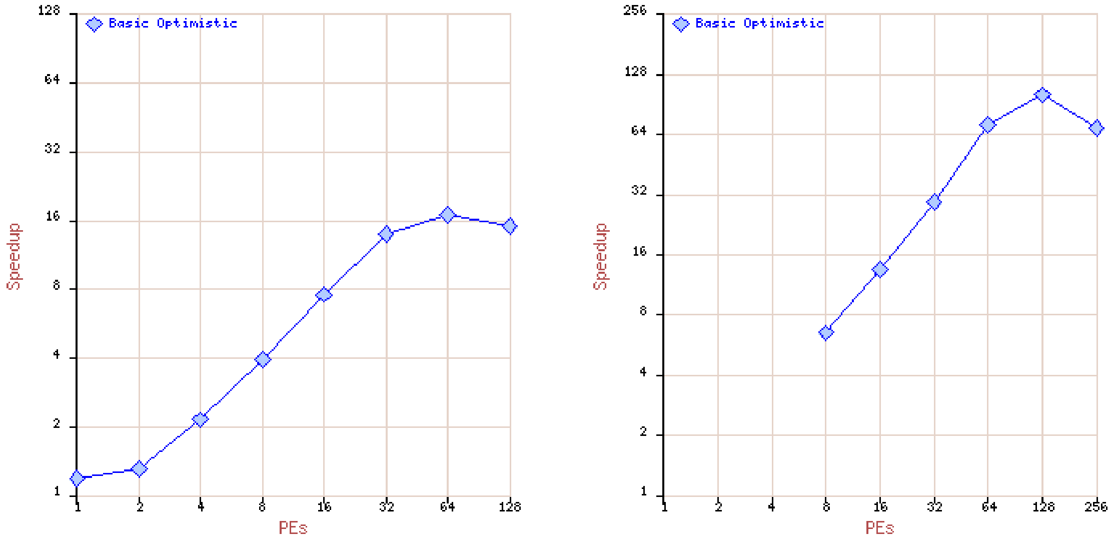


Figure 6.2: Optimistic Speedup for small (left) and large (right) instances of synthetic benchmark

The sequential code experienced some memory thrashing for the large problem instance that possibly lead to inflated execution time. To avoid the measurement artifacts associated with this, we calculated $T_e = 2025.3$ from the number of events executed multiplied by a per event synchronization overhead plus the forward execution time. Our speedup graphs for the large benchmark are thus plotted relative to T_e.

These first experiments illustrated both the problem with sequential simulation as well as the need to improve parallel performance. The sequential simulation was not able to run efficiently in the available memory. Parallel simulation solved this and obtained much faster times; however the performance of the Optimistic strategy quickly degraded as work per processor declined. This behavior is largely due to excessive speculative computation resulting in rollbacks and cancellations. The object model localizes the extra overhead for rollback and cancellation effectively, but it cannot reduce the extra forward execution time accrued simply from executing so many useless events in spite of their fine granularity.

Table 6.1 shows[1] how much extra work Optimistic performed on the small benchmark as the number of processors increased. The effect is even more profound for the larger benchmark with higher event density: 57.6% of events executed were useful on 128 processors.

Processors	Total Events	Useful Events	% Useful
1	3229933	3229933	100%
2	3229465	3229465	100%
4	3237226	3227515	99.7%
8	3452537	3228123	93.5%
16	3551278	3228112	90.9%
32	3923574	3229102	82.3%
64	4415306	3227589	73.1%
128	4695925	3230797	68.8%

Table 6.1: Speculation in the small benchmark increases with number of processors in the Optimistic strategy

Because events in Optimistic are executed without regard for how far ahead of the GVT they are timestamped, there are no controls on speculation. The speculative work keeps the simulation busy and free of idle time. However, the GVT algorithm is invoked infrequently as it relies on natural idle periods in which to perform its function (more on this in Chapter 7). This means that fossil collection rarely takes place and memory usage is excessive. The GVT algorithm is designed to work better when its data structures are small. No matter how frequently the GVT computation is invoked, it still needs to account for all the activity in the simulation from the previous GVT estimation up to the point at which it is invoked again. Running it infrequently results in long laborious processing of large data structures when it finally does run. Thus GVT estimation is the dominating overhead component in Optimistic on fewer processors. This is shown for the large benchmark in Table 6.2. Times indicated are average parallel execution time per processor in seconds. Note that the GVT overhead scales superlinearly because the size of the local PVT data structures on each

[1] Note that the slight variation in number of events executed is due to the termination method for the benchmark. It terminates when a preselected endtime t_{end} is reached. As the number of processors varies, the message to terminate reaches each processor at different points in time. Thus some additional events may be executed before each processor realizes the termination condition has been reached.

processor decrease as the number of processors increases.

Procs.	Forward Exec.	GVT	Sync.	Fossil Collect	Checkpoint	Rollback	Cancel	Comm. & System
8	41.94	100.13	4.78	6.70	1.67	0.25	0.08	13.51
16	26.58	32.77	2.45	2.56	0.86	0.19	0.13	3.32
32	12.44	9.65	1.00	1.27	0.38	0.06	0.01	1.81
64	5.67	3.76	0.56	0.58	0.22	0.08	0.06	1.87
128	2.74	1.99	0.27	0.32	0.08	0.06	0.04	6.22
256	1.79	0.75	0.16	0.13	0.07	0.06	0.07	15.13

Table 6.2: GVT dominates Optimistic average parallel execution time in large benchmark

Another significant source of overhead comes from the synchronization activities. Each time there is a chance of performing any computation on an object, it needs to go through the motions described earlier: possibly perform fossil collection, check for cancellations and handle them, check for stragglers and handle them and finally forward execution. In Optimistic, this happens once for every event that gets executed. Our next strategy, Batched Optimistic, addresses this issue.

6.1.2 The Batched Optimistc Strategy: Multi-events

When an object gets control, it may have more than one event in its queue. A single rollback could undo several events, but the one-at-a-time nature of Optimistic would re-execute them slowly, interleaving with other object's events. Batched Optimistic allows for the processing of several events in a row, *i.e.* as a *multi-event*. A multi-event is simply a sequence of events that are executed on a single poser with no intervening overhead for scheduling, rollbacks, cancellations, GVT tasks or any other kind of overhead performed in optimistic synchronization. In the case of Batched Optimistic, we require that the events comprising a multi-event have the same timestamp. This preserves the best-first ordering of Optimistic, but reduces the total number of iterations through the synchronization mechanism. It produces similar performance results to Optimistic as shown in Figure 6.3, but did achieve a larger *effective grainsize* which we will discuss shortly. One difference we noted on several runs was that the synchronization overhead for Batched Optimistic was often less than that for Optimistic. It was not significant enough, however, to improve speedup, mostly because the number of multi-events that could be formed were limited due to the restriction that the events comprising the multi-event should have the same timestamp (see Table 6.3).

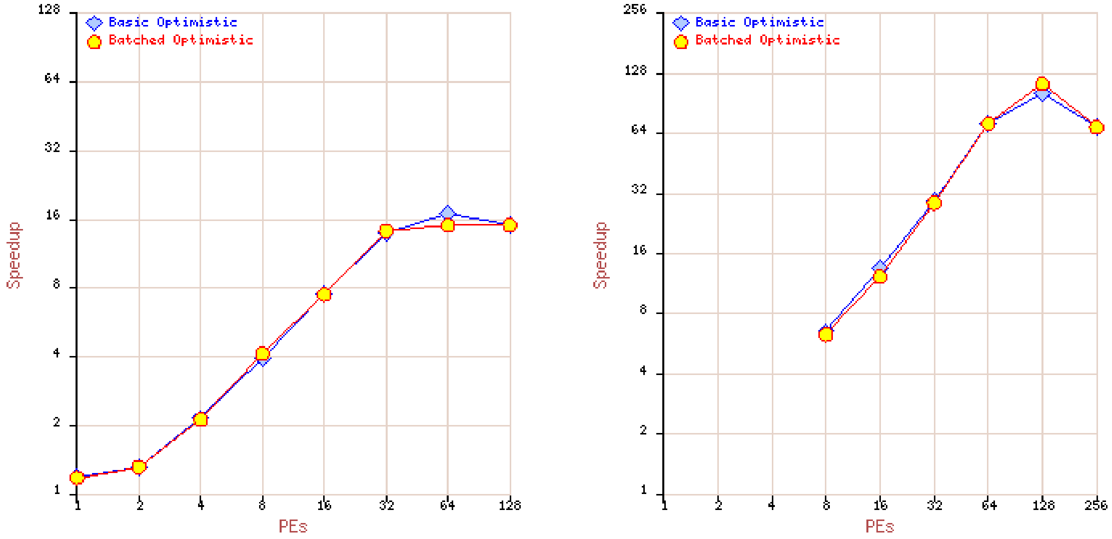


Figure 6.3: Batched Optimistic Speedup for small (left) and large (right) instances of synthetic benchmark

6.1.3 The Throttled Optimistic Strategy: Time Windows

Given the significant overhead of the GVT component (see Table 5.2), we wanted to improve Batched Optimistic by having the GVT calculated more frequently on smaller data structures. To do this, we added a time window to the strategy. The time window amounts to a "leash" of a certain length placed on a poser's synchronization strategy. The window is a range of virtual time starting at the GVT and extending some w time units into the future. When an object gets control, it can only execute its earliest event if that event is timestamped with a time that is between GVT and GVT+w.

Depending on the size of time window chosen, Throttled Optimistic exhibits much lower GVT and rollback overhead than the earlier optimistic strategies, but takes more time for synchronization. This is because the synchronization mechanism is invoked many more times than before, since some of those iterations may now result in no forward execution at all. The new effect is significantly poorer performance for fewer processors, but a slightly better ability to scale as the number of processors increases. In addition, calling the GVT more frequently allows fossil collection to run more often, thus freeing up memory that is no longer needed. We see this effect by the fact that we were able to get a run completed on 4 processors for Throttled Optimistic that would run out of memory for Optimistic and Batched Optimistic. The comparison to Optimistic is shown in Figure 6.4. The choice of time window affects the performance on Throttled Optimistic dramatically, with a trade-off between the types of overhead. Our experiments were performed with a value that produced best overall speedup on large numbers of processors, determined via trial and error. This window size was then used for all the runs. Larger time windows reduce the synchronization cost, but increase the GVT cost, while smaller windows have the opposite effect. The effect on the GVT overhead is shown in Table 7.1 in the next chapter. There we compare the GVT overhead of Optimistic with that of Adaptive (which behaves similarly to Throttled Optimistic with respect to GVT overhead). The most significant difference between Throttled Optimistic and Basic Optimistic is in the forward execution and synchronization behavior. This is greatly improved as we shall see in Table 6.3.

Although the improvements of the Throttled Optimistic strategy are overwhelmed by the problems discussed earlier, we shall expand and combine the techniques of our optimistic strategies to make more effective approaches. With our three basic optimistic strategies, we learned how certain behaviors could be adjusted to have effects on certain overhead components. These observations have been used to design our speculative and adaptive strategies as described below.

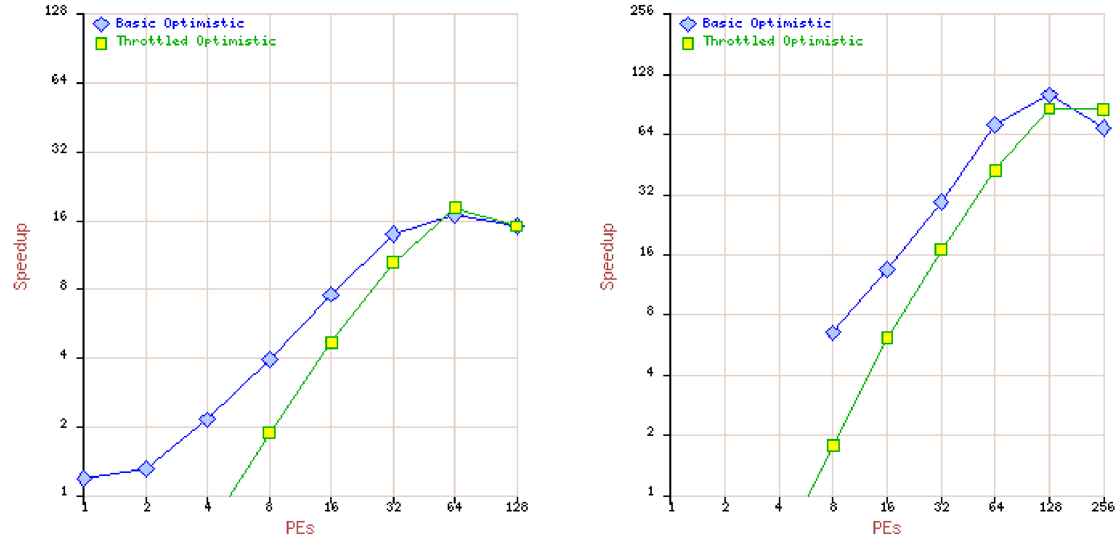


Figure 6.4: Throttled Optimistic Speedup for small (left) and large (right) instances of synthetic benchmark

6.2 Speculative Synchronization

In speculative synchronization, we try to maximize the benefits of having a time window by allowing more events that fall within the window to be executed as a multi-event with no intervening overhead. We discussed speculative computation briefly in Chapter 2. All optimistic strategies perform some amount of speculative computation. In our basic optimistic approaches, an event arrives and is sorted into the processor-level event list and then we take the earliest event and execute it. Though we know the event is the earliest available on the processor, we do not know if it is the earliest in the entire simulation. Thus, executing it still involves a bit of speculation.

In the speculative approach, the behavior is similar, with some exceptions. First, the time window is replaced by a *speculative window* that governs how far into the future beyond the GVT estimate an object may speculate. Speculative windows are similar to the time windows[42] of other optimistic strategies including our own variants. The key difference is in how events within the speculative window are executed on a per object basis.

As described earlier, the earliest event in the Charm++ scheduler is first inserted into the event queue on the object it is destined for. When the object invokes the speculative strategy to process events, the event just received will often be the next event processed and no other events will exist on that object. However, in some cases, there may be several events queued on a poser. This can happen for example when we rollback several events, or when events with timestamps beyond the time window were queued but not executed. The earliest of these events may be worthy candidates for speculative execution. Others may have timestamps very far in the future and it may be unwise to execute them speculatively.

Our strategy must determine how to process the events within a speculative window. The natural approach is to schedule objects according to the timestamp of their earliest event, but we have seen from our basic optimistic strategies that this approach has a rather high overhead associated with each event due to the additional scheduling and context switching involved. Executing batches of events at one time reduced this overhead, as seen in Batched Optimistic.

To enhance this effect, the speculative approach allows objects more freedom to speculate once they get control. Event processing on the object now follows this pattern: if there are events with timestamp within the speculative window, execute *all* of them. The later events are probably not the earliest in the simulation and it is likely that they are not even the earliest on that processor. Thus we are no longer executing events in best-first order on a processor. We are allowing the strategy to speculate that those events are the earliest that the *object* will receive and thus represent a best-first ordering of events on the *object*. By handling events in batches, we reduce scheduling and context switching overhead and benefit from a warmed cache, but risk additional rollback overhead. However, since we are still working within a window, we could still be performing less speculation than Optimistic or Batched Optimistic which may speculate as far into the future as they please.

In the optimistic strategies, each time an event is executed a token message is enqueued in the Charm++ scheduler if there are additional events remaining on the poser, resulting in one invocation of the synchronization mechanism per event. In the speculative approach, we no longer need to rely on token messages to execute remaining events on an object. Event arrivals and increases in the GVT estimate will prompt each object to perform forward execution only when necessary. This means we invoke the synchronization mechanism a maximum of once per executed event in a simulation, but on average it is invoked much less frequently.

Speculation brings about a number of interesting performance side effects. As in the Throttled Optimistic strategy, the limitations of the speculative window allow for idle periods (where a processor runs out of speculative work to do within the speculative window) during which the GVT calculation can occur. More frequent GVT calculations on smaller sets of data are much faster than less frequent calculations on larger data as happens in the Optimistic strategy. This is because the range of virtual time represented is smaller which allows for more consolidation in the GVT data structures. This makes the data structures smaller and the operations on them faster.

Since we never reschedule events on an object, we avoid the overhead of rescheduling events in the Charm++ scheduler as well as the overhead of returning to the object frequently to execute events. In Table 6.3 we show how speculation combines several event executions into one *multi-event*, thereby increasing the *effective grainsize* of a simulation. The effective grainsize is the total time spent executing all events divided by the number of multi-events. For this experiment we ran a very large instance of the synthetic benchmark on 4 processors. We begin to see the effect in the Batched Optimistic and Throttled Optimistic strategies which execute available events with the same timestamp as a multi-event, but Speculative achieves a more dramatic improvement than the earlier approaches.

Another interesting side effect that occurs in some situations is a reduction in synchronization time and possibly grainsize. We compared a sequential POSE run of our synthetic benchmark to two single-processor parallel POSE runs, one with the Batched Optimistic strategy and one with an Adaptive speculative strategy and discovered that the time to synchronize events was dramatically reduced in the parallel approaches (see Table 6.4). We

Strategy	Forward Execution	Events	Speculative Events	Grainsize	Multi-Events	Effective Grainsize
Optimistic	581.34	13671247	41515400	14μs	41515400	14μs
Batched Optimistic	518.81	13671247	39904246	13μs	37317020	14μs
Throttled Optimistic	205.07	13671247	13671252	15μs	12311015	17μs
Speculative	205.03	13671247	13671249	15μs	8204482	24μs

Table 6.3: Multi-events and effective grainsize

theorized that this improvement was due to subtle event re-orderings that allowed the parallel strategies to execute multi-events on single objects which in turn would avoid context switching and improve cache performance. Of course in the parallel approaches we add a lot of other overhead, for example from the GVT component as shown here, as well as checkpointing, fossil collection, etc., which unfortunately easily make up for the saved time.

Strategy	Execution Time	Forward Execution	Synchronization	GVT
Sequential	78.808	12.88	60.36	0.00
Batched Optimistic	66.932	13.81	6.14	40.98
Adaptive	69.508	13.84	6.18	44.16

Table 6.4: Sequential vs. Batched Optimistic vs. Adaptive

We made our design decisions so that processing events on a single object would be “more like sequential”, but what we ended up with was better than sequential. In fact, early versions of the speculative strategies with earlier versions of the synthetic benchmark (which produced a higher density of events per virtual time unit) exhibit very interesting cache behavior compared to sequential simulation. It appears that the improved cache performance is due to the ordering of events allowing them to be executed in a warmed cache. To verify this, we ran a simulation with Lemieux’s atom tool for performance analysis and obtained the cache performance results shown in Table 6.5.

Not only were there fewer cache misses, but due to all the other activities performed in the parallel case the total number of references was much higher, leading to a dramatically better hit ratio for the parallel speculative strategies than for sequential. We have since chosen to

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

Table 6.5: Caching behavior of sequential vs. parallel POSE

implement most of the speculative strategies to perform more “conservative” speculation, and thus the caching behavior has changed for fewer processors as we see in Figure 6.5.

We compare our Speculative strategy to Optimistic in Figure 6.5. The Speculative strategy further enhances the effect of Throttled Optimistic, obtaining slightly better speedups on larger numbers of processors. It still performs poorly on lower numbers of processors, but is slightly better than Throttled Optimistic.

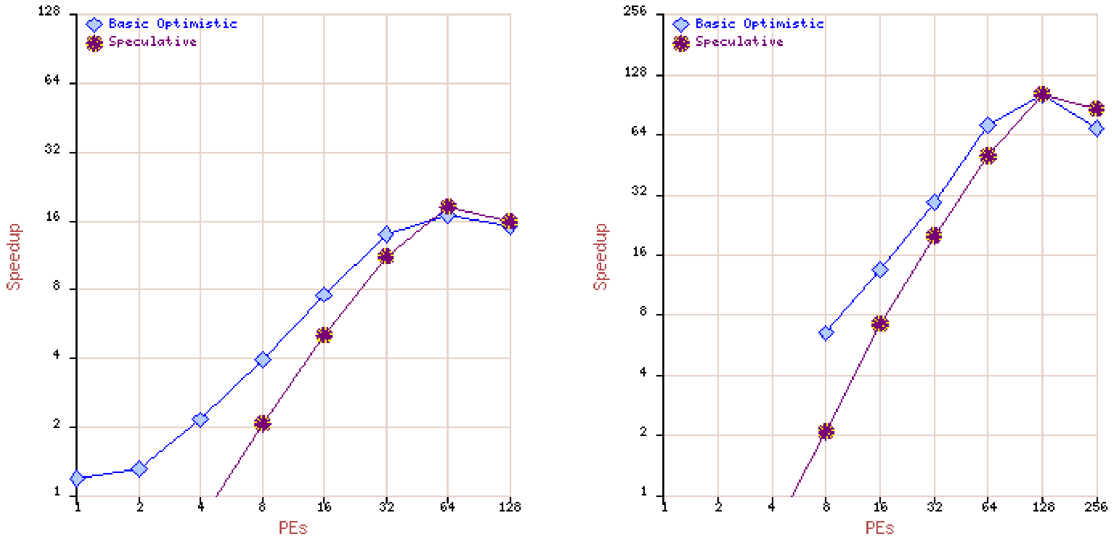


Figure 6.5: Speculative Speedup for small (left) and large (right) instances of synthetic benchmark

Speculative allows for the execution of events within the speculative time window as a single multi-event. The window size is fixed and must be chosen to be suitable to an application. Too large a window could overwhelm the simulation with rollback and cancellation time. Too small and the simulation might be constantly shifting from one object to the next, calling the GVT very often in between. We shall see how we can better control the window size in the next section.

The speculative strategy, however, could lead to expensive rollback situations. We need to exploit the benefit of speculation as much as possible without letting it get out of control. Although the limitations provided by the speculative window aid in this, we still have the problem leftover from Throttled Optimistic: choosing the window size is challenging and different window sizes lead to different performance for different applications. Thus we need some way to automatically control the size of the window. This idea forms the basis for the Adaptive synchronization strategy discussed in the next section.

6.3 Adaptive Synchronization

So far we have seen our strategies evolve from the optimistic extreme in Optimistic, to a slightly more conservative approach with time windows in Throttled Optimistic, to a hybrid of time windows and a new sort of extreme optimism in the form of the multi-event in Speculative. We have seen firsthand the trade-off between optimism and conservatism in terms of cost: lost opportunity for parallelism in the time window case and rollbacks and associated costs when no such limits are in place (see Figure 6.6). We have also seen that the impact of the trade-off varies depending not only on attributes of the application, but also on the environment in which the application executes, particularly concerning the number of processors used.

As we see in Figure 6.6, there is a sweet spot between the conservative and optimistic extremes which many variations of traditional strategies have tried to achieve.

For example, our Speculative strategy has a speculative window which effectively allows speculation while keeping rollbacks under control. But what size should that window be? We experimented with different sizes and discovered that some simulations benefited from a very small window, while others could run fastest with a huge window and still not rollback. There are several problems with having one window to rule all objects and all simulations.

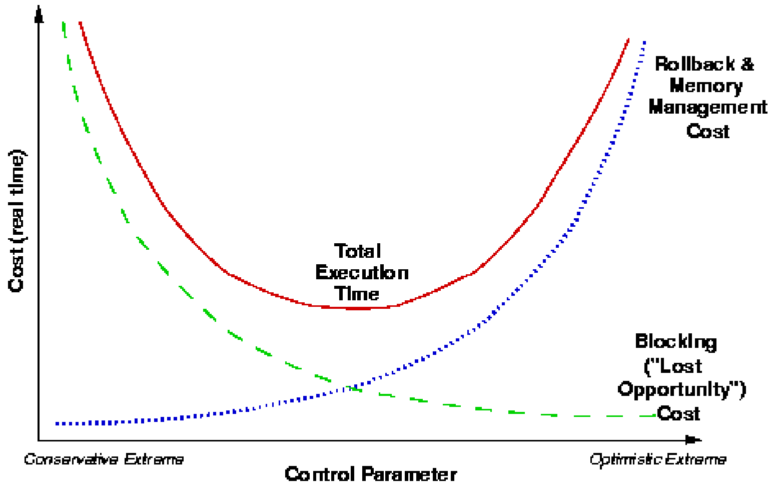


Figure 6.6: Trade-off between conservative and optimistic protocols (from [7])

Application Behavior There is no one speculative window size that will work best for all applications.

Application Variance There is no one speculative window size that will work best for the entire run of an application with varying behavior.

Entity Behaviors There is no one speculative window size that will work best for all the posers in a simulation.

Entity Variance There is no one speculative window size that will work best for a single poser with varying behavior.

Clearly, strategies need to be able to adapt to ever-changing situations [7]. The solution we propose is enabled by the fine-grained objects of Pose. In this solution, each entity is created with its own instance of an adaptive strategy which adapts the speculation on the object to the object's past behavior, current state and likely future behavior (given what future events and cancellations are queued on the object).

We developed an Adaptive strategy in Pose that keeps track of past behavior by monitoring how much speculation the object has "gotten away with" without adverse effects. When too much speculation is occurring, the strategy reduces the speculative window size and restricts its ability to expand. When the object speculates successfully, the window is expanded, but not beyond what is known about the object's future according to the events queued on the object.

We have implemented a variety of adaptive strategies that combine the techniques of the previous strategies with the ability to adjust the speculative window size on a per object basis.

The first generation of the Adaptive strategy allowed each object to have its own speculative window and "punished" objects that rolled back by shrinking their windows to some minimum, while it "rewarded" objects each time they successfully processed events by letting them gradually expand their windows up to some maximum. We compare this Coarse Adaptive strategy to Optimistic and Speculative in Figure 6.7.

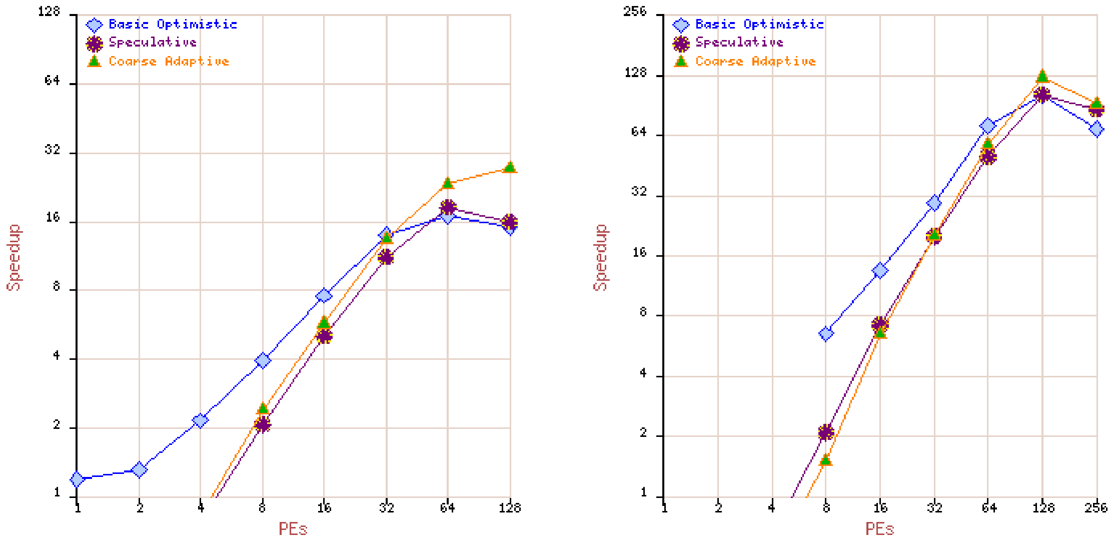


Figure 6.7: Coarse Adaptive speedup for small (left) and large (right) instances of synthetic benchmark

The performance is similar to Speculative until we start to run out of work to do on larger numbers of processors. Then Coarse Adaptive becomes slightly more adept at handling speculation. For example, on 64 processors for the small benchmark, both strategies are executing 100% useful events. Coarse Adaptive however is executing much larger multi-events. Speculative forms 2264475 multi-events out of 3219589 total events, whereas Coarse Adaptive forms 960595 out of the same 3219589 total events. Thus, multi-events have an average size of 1.4 events for Speculative, while the average is 3.4 events per multi-event for Coarse Adaptive. The ability to adapt speculation seems key as the number of processors increase. We designed our next generation Adaptive strategy to be more finely tuned to the range in which an object can safely speculate.

Fine Adaptive monitors how much virtual time rollbacks undo and restricts the window to be within this range. For example, if a simulation runs for a time without any rollbacks, the window gradually expands. If a straggler is received, we shrink the window to a size that includes just the straggler, but no later events. The window size can be expanded from there. Thus, the "punishment" for rollback is less severe than it was in the earlier version. The speedup of Fine Adaptive is compared to the Coarse Adaptive and Optimistic in Figure 6.8.

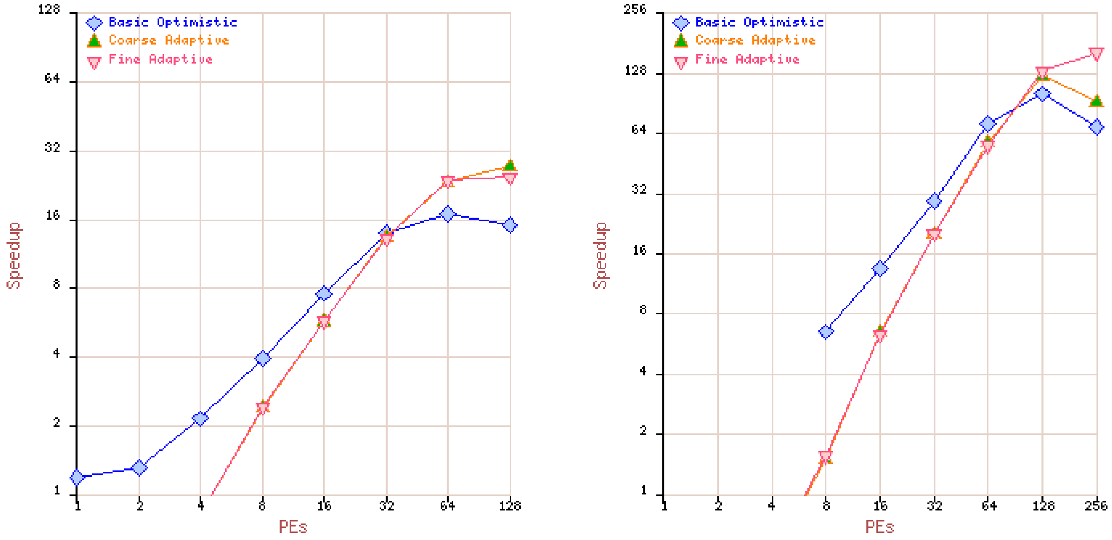


Figure 6.8: Fine Adaptive speedup for small (left) and large (right) instances of synthetic benchmark

Having a less severe, more specific punishment allows the speculative window to shrink only as much as necessary. It does not take as long to expand as the poser realizes speculation is possible. This again allows for the formation of larger multi-events and slightly improves the performance over the earlier strategy for the larger numbers of processors. For the small benchmark on 64 processors, Fine Adaptive now forms only 763194 multi-events to 3219589 total events, making for an average multi-event size of 4.2 events.

Ideally, our desired speedup performance for our final Adaptive strategy should appear in Figure 6.8 as a curve that skims over the top of all the curves shown there. Many questions arose as to how to determine when our new Adaptive strategy should behave like Optimistic and when should it behave like the previous Adaptive strategy. One short-lived variant performed very well when it behaved one way or the other according to the number of processors it was running on. However, determining at what number of processors to switch from one behavior to the other varied from application to application, and discovering the ideal value for a single application was only possible through trial and error. This definitely took away from the idea of adaptivity of our strategies. Thus we needed to find a key behavior (or set of behaviors) to examine that would allow each poser to adapt its speculative window to achieve to desired behavior.

We made the following key observations:

1. On very few processors, the percentage of useful events is high and rollbacks and cancellations are rare. This is true for all the strategies. Thus the limits of the speculative window provide no benefit since rollback is not likely.

2. The speculative window becomes important when we need to restrict speculation on higher numbers of processors. This is where the trade off occurs: too large a speculative window leads to more rollbacks, but too small restricts the size of multi-events and thus the amount of synchronization overhead that can be avoided. Thus, adaptivity at this point is crucial.

3. The crossover point (when limiting speculation becomes important) can be determined by how successfully we are speculating with *no limits* on the size of the speculative

window, i.e. given free rein, how much of the work done so far is useful. If useless work is in excess, we need to impose limits.

To incorporate these observations into the Adaptive strategy, we added a measure of *speculative tolerance* which specifies a maximum percentage of the events that we allow that are not useful (*i.e.* events that get rolled back). Speculative tolerance is set at 10%, i.e. an object will continue to speculate as long as the number of non-useful events remains under 10% of the events executed *on that object*. We start out with a very large window which is set to cover half of the entire possible virtual time range. This places our strategy far out beyond the extreme of "extreme optimism" due to the presence of multi-events which allow us to violate best-first ordering on a processor. Our strategy proceeds to speculate based on available work and always tries to maximize speculation. Punishment occurs on an object when it exceeds its speculative tolerance or when a rollback occurs. In the case of a rollback, we use the same fine-tuned punishment scheme of the previous version. If we have exceeded the speculative tolerance, we reduce the window size to be within the range where rollbacks do not occur on average. Then speculation is limited to only the earliest work available. When enough non-speculative work is completed such that the percentage of speculative work is under the tolerance again, more speculative work will then be allowed. Successful window expansions are rewarded with more window expansions. The quanta used for expansion and contraction of the window is determined by the average rollback offset. In other words, an object that frequently rolls back to a time close to the GVT is only allowed to change its speculative window size by small amounts at a time, whereas an object that rolls back to points far beyond the GVT can expand and contract its window much faster. There are no minimum or maximum window sizes imposed on the posers, thus this strategy is the most flexible at adapting to widely varying situations.

Thus, we have a strategy that starts with a huge window, and if it does not encounter problems with rollbacks, as is the case for fewer processors, it proceeds in much the same way as Optimistic and Batched Optimistic. When we increase the number of processors and rollbacks start to occur, the window size is adapted to deal with this.

Figure 6.9 displays the range in which Adaptive operates. It also shows how we have reduced the overhead of optimistic synchronization via virtualization and the tiny LP model, and stretched the extreme of optimism with additional speculation via multi-events.

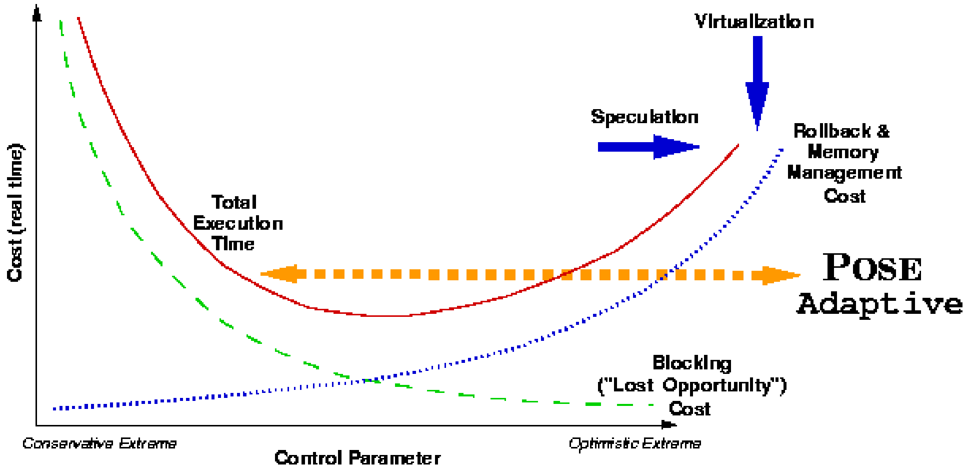


Figure 6.9: The reduced cost of extreme optimism in POSE and the adaptive range of the Adaptive strategy

We compare the new Adaptive strategy to the previous version and Optimistic in Figure 6.10. While the earlier version shows poor performance for fewer processors, it obtains the best speedups of all strategies on larger numbers of processors. Our best speedup for the large benchmark was achieved by the earlier version and is approximately 180 on 256 processors. The peak efficiency speedup was also achieved by the earlier Adaptive strategy at about 100% on 128 processors. The new Adaptive strategy managed to compete very well with Optimistic on fewer processors. On higher numbers of processors, it performed much better than Optimistic, but not quite as well as the earlier version of Adaptive. Figure 6.10 shows the new Adaptive curve coming close to our goal of skimming the top of the other two curves, but there is room for improvement on large numbers of processors.

The new Adaptive strategy also falls short of our goal of being at least as memory efficient as the previous version. The problem lies with the fact that the GVT estimation is only invoked in idle periods of time. As in Optimistic, the new Adaptive strategy invokes the

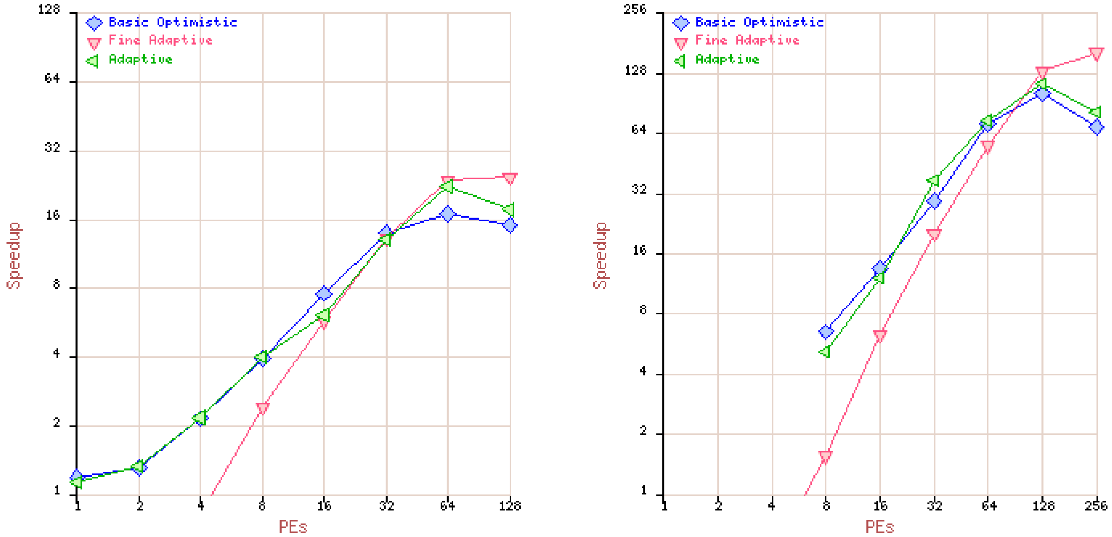


Figure 6.10: New Adaptive Speedup for small (left) and large (right) instances of synthetic benchmark

GVT only a few times when executing on fewer processors. In the future, we will add the ability to invoke the GVT when too much memory is in use. Since all the strategies use significant quantities of memory, we have implemented an extension that reduces memory usage by limiting speculation when memory usage is high, though this comes at the cost of performance. We also have an event message recycling facility in place which reduces allocation and deallocation costs. We plan to add a general memory management facility for POSE as well.

6.4 The Adept Synchronization Strategy

Our final strategy is intended to be a culmination of the knowledge gained from the previous strategies. It has existed in many variants, a few of which we shall discuss here. It started as a true culmination of the ideas in the previous strategies, but did not amount to much. Our goals were 1) to achieve the peak efficiency speedups of an Adaptive strategy, while 2) obtaining similarly efficient performance to Optimistic on fewer processors in 3) a memory-efficient manner.

The Adept strategy is the final Pose synchronization strategy that we will discuss. It adapts on a variety of levels to a variety of situations. It may adapt to other behviors in the future. It is the single unified strategy meant to be used as the default strategy in Pose for handling general-purpose PDES.

Adept first makes use of the unlimited speculation of the Adaptive strategy. It uses a fine-tuned punishment/reward mechanism for shrinking the speculative window in response to rollbacks and expanding it when forward execution is successful. It incorporates a speculative tolerance at both object and processor level, limiting the actual *number* of potentially useless events it can execute in a single multi-event. This is particularly useful when huge numbers of events around the same timestap fall within the speculative window. The object will execute some of them, and then give up control so that other objects can execute and stragglers have a chance to arrive in the meantime. In addition, it imposes an adaptive memory management scheme at object and processor level. Objects with a large quantity of checkpointed state are throttled temporarily until and updated GVT invokes fossil collection to occur. This both reduces memory usage and speeds up the operations on the GVT data structures.

An early version of Adept was implemented similarly to Adaptive, but without memory management. This performed quite well as we see in Figure 6.11.

The strategy in its nearly completed form has much of the final adaptive abilities that we desire, with the exception of more processor- and global-level controls on speculation ranges in virtual time, speculative event quantities and memory management.

We show some preliminary performance results for this strategy. Our first experiment with an updated version of the synthetic benchmark indicated that Adept performed nearly identically to Optimistic under ideal (infrequent rollback, small problem size) conditions. Adept suffered a small performance hit for synchronization but ultimately performed with similar speedup to Optimistic on fewer processors, and slightly better on more processors. However, under less ideal conditions with more likelihood of rollback, high memory usage and high message density, for example, Adept is far superior at adapting to the problem. We

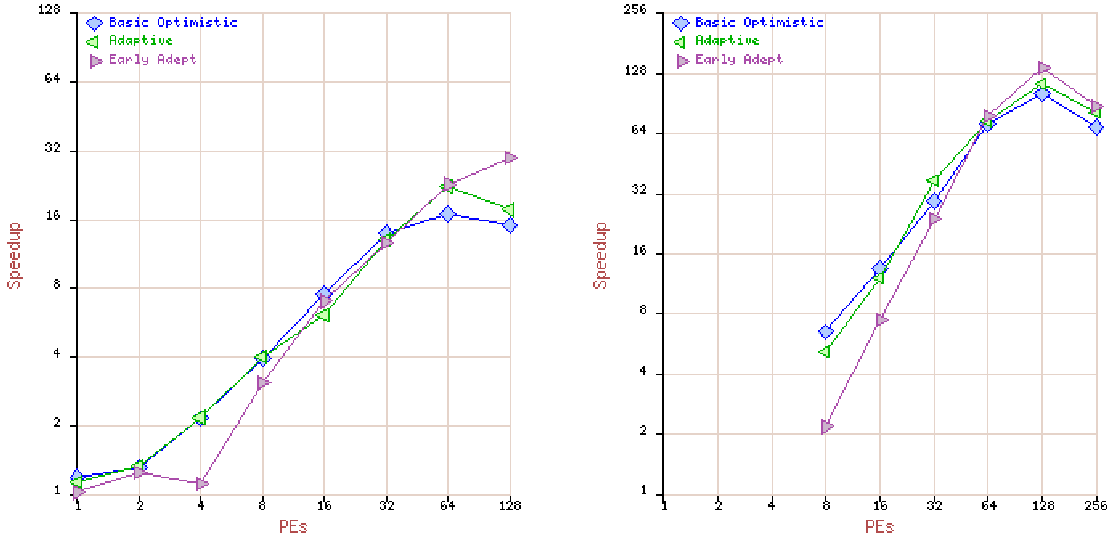


Figure 6.11: Early Adept speedup for small (left) and large (right) instances of synthetic benchmark

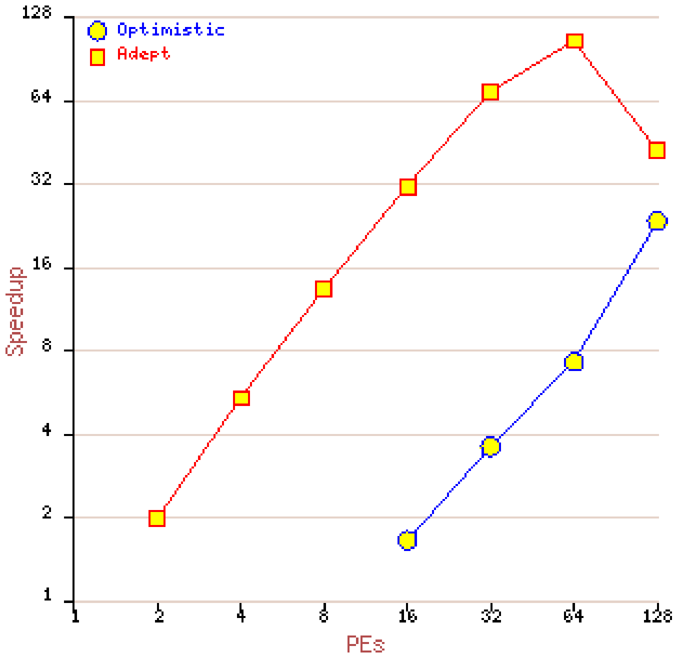


Figure 6.12: Adept vs. Optimistic for Multiple Rings benchmark

show a Turing run of Adept vs. Optimistic in Figure 6.12 on a Multiple Rings benchmark. This program implements a set of overlapping rings of events, one started by each worker object. In this run, there are 10240 objects and an event density of 20 events per virtual time unit. The program terminates when all objects have each received 2000 events. Thus the total number of events handled is 20,480,000. The average grainsize of the events is 5 microseconds on Turing. Since the sequential run of this benchmark ran out of memory, we used the 2 processor time for Adept as our base sequential time to illustrate the speedup. As the figure illustrates, Optimistic is overwhelmed on fewer processors, and was unable to complete the run due to insufficient memory until the 16 processor experiment. Adept was able to complete a run on 2 processors, and achieve excellent speedup up to 64 procs. The superlinearity of the curves is due to the decrease in the impact of insufficient memory as the number of processors increases. At 64 processors the program completed in 5 seconds and could not speedup beyond that. It should be noted that the previous Adaptive version used too much memory to complete any of the runs up to 128 processors.

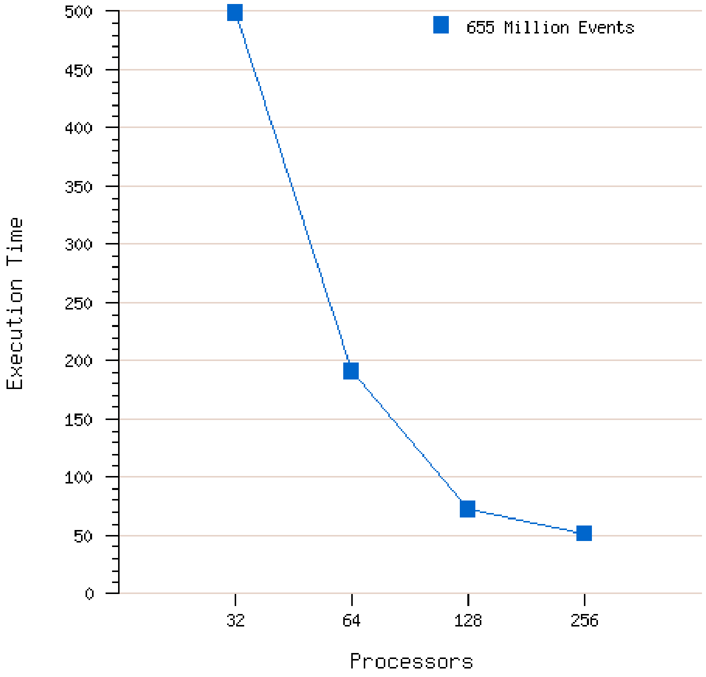


Figure 6.13: Scaling of Adept with large instance of Multiple Rings benchmark

Another experiment shows how huge problem sizes scale with Adept. Figure 6.13 shows execution times between 32 and 256 processors on Turing for the Multiple Rings benchmark, performing a total of 655,360,000 events. Superlinear speedups are due to memory availability. The program has difficulty scaling to 256 processors but we were unable to get such runs completed with any sort of frequency. We plan to analyze this problem further in the future when we have better access to large parallel machines.

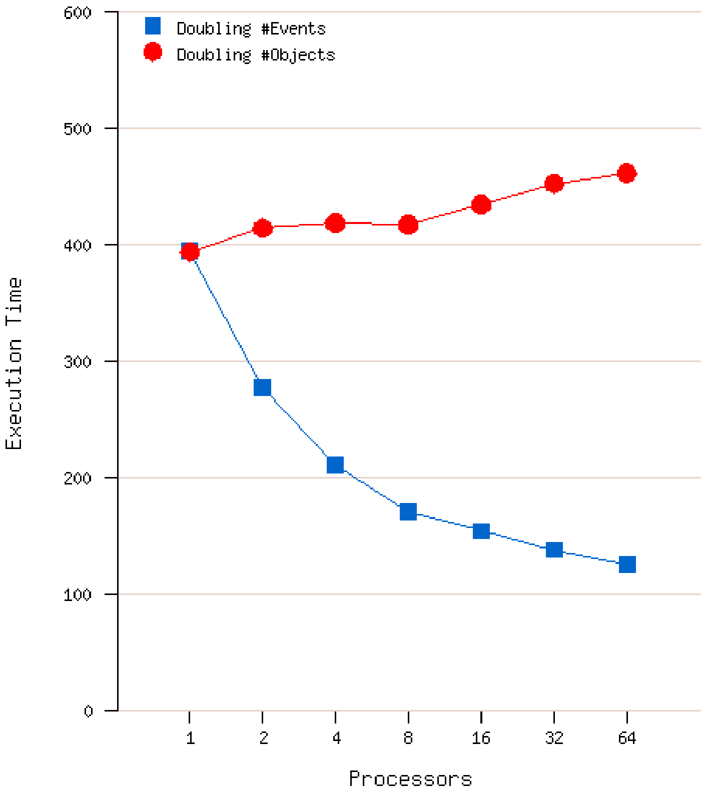


Figure 6.14: Problem size scaling of Adept with Multiple Rings benchmark

As our final strategy for Pose, Adept exhibits excellent scalability. Figure 6.14 shows how Pose using Adept scales with problem size, again using the Multiple Rings benchmark. As we double the problem size, the event density also doubles. In the case of problem size doubling by doubling the number of events handled per object, we see that doubling processors results in a decrease in the time taken to execute the double-size simulation. This is largely due to the aggressive amounts of speculation using multi-events. More events can thus be handled in less time reducing the amount of synchronization overhead and improving cache performance. Doubling problem size via doubling the number of objects has quite a different result. The time gradually increases as processors and objects double. This is largely due to the time taken to construct the additional objects on the additional processors, but also includes extra overhead for object management in the various data structures. Since the number of events per objects remains constant in all the runs, the size of multi-events does not increase.

6.5 Summary

Pose's object model based on virtualization provide the groundwork for scalable optimistically synchronized PDES by localizing overhead to small entities. In this chapter, we developed adaptive speculative synchronization strategies that leverage the fine grainsize of entities to adjust their synchronization behavior appropriately. In Figure 6.15 we show the average overhead per processor for the large benchmark and how it scales for our Adaptive strategy. Sequential time (not plotted on the graph) was $2824.8s$ but we used an estimate of $1412.4s$ instead in our speedup graphs in the event that the sequential simulation was thrashing in inadequate memory. We see from the overhead chart that in Adaptive most overhead scales very well. Space between the bar and the total parallel execution time curve indicates how much average differs from maximum time on a processor. The larger gaps indicate load imbalance.

As of this writing, Adaptive exhibits much of the sort of behavior we would like from an adaptive strategy. We show an overview of a very small synthetic benchmark run on 16 processors in Figure 6.16. The overview graph has time on the x-axis and processors on the y-axis. The left graph shows utilization per processor over time, and the right graph shows active Charm++ entry points over time. For the utilization graph, brightness or intensity of color indicates higher utilization. This graph shows a long period of intense

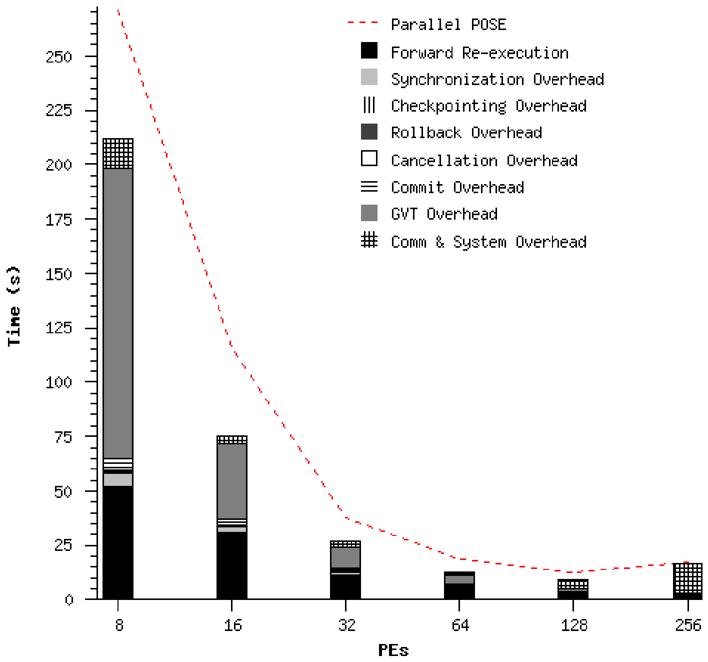
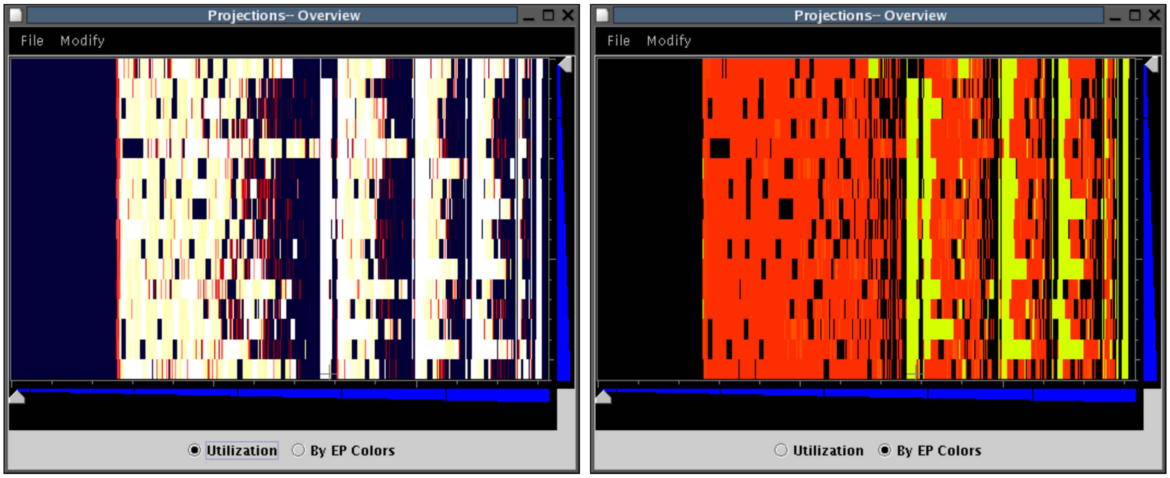


Figure 6.15: Adaptive overhead for large problem instance

computation followed by scattered idle time on the processors. Then the intensity starts up again. This coincides with the reporting of a new GVT estimate. The simulation proceeds with denser more intense computation and the frequency of the GVT increases because speculative windows have reduced in size. The right graph colored by entry points shows the GVT operations in the palest (yellow) color and simulation events handled in the darker grey (red). Note that much of the pale color is also event handling because the code that is invoked to set the new GVT on the processor also checks each poser for newly enabled work. We discuss the GVT behavior in this particular run again in the next chapter.

We have focused our efforts on a synthetic benchmark which exhibits fine granularity and achieves near-linear speedups. However, this benchmark also exhibits a high degree of parallelism — many objects per processor have work at any given time. We also simulate a constant behavior for a period of time, so the degree of parallelism remains high throughout the simulation. For such simulations, POSE enables optimal performance. The Multiple Rings benchmark exhibits high communication, high memory usage and higher event density.



● Utilization ○ By EP Colors

○ Utilization ● By EP Colors

Figure 6.16: Small benchmark overviews showing utilization and entry points

It is more likely to rollback with Optimistic synchronization. The Adept strategy handles these challenges well and far better than our previous Adaptive approaches. Chapter 11 addresses the challenges of a system model where the degree of parallelism is relatively low on average.

Chapter 7

Global Virtual Time in Pose

Global Virtual Time (GVT) is the maximum time that all state has progressed past in the entire simulation. Calculating it allows us to know which checkpoints of past events are no longer needed and can be freed. As we have seen, it can be used in a number of other ways, such as an indication of simulation progress and a means for determining if the simulation has reached a termination state. Calculating the GVT is a complex, time-consuming task that we need to pay considerable attention to when studying scalability of PDES.

7.1 Our requirements of a GVT algorithm

The design of our GVT algorithm evolved along with the design of our synchronization strategies. As the synchronization behavior changed, we found we had more requirements of the GVT algorithm, which put further burden (in the form of overhead) on this component of our PDES environment.

We started with a simple synchronous approach that accompanied the first optimistic synchronization strategy. GVT calculation was periodically invoked and involved all processors reaching a barrier and detecting quiescence to make sure no events or cancellations were in transit. They would then report their *processor virtual times* (PVTs) to a central processor which would minimize over all the PVTs to obtain the GVT. This worked fine enabling us to obtain an accurate GVT and perform fossil collection. The biggest problem was figuring out how frequently it should run. Various trade-offs exist. Simulations pay a higher cost in idle time and overhead the more frequently the GVT is invoked. Invoking it less frequently results in higher memory usage for the storage of checkpoints which was already excessive from the start.

Since uncontrollable rollbacks were an issue in the basic optimistic strategy as the number of processors in the simulation increased, we knew we would be looking into the use of time windows. This creates a dependency between the synchronization strategy and the GVT which may seem problematic but which turns outs to be an opportunity in our adaptive approaches.

To achieve good progress with a time window strategy, we need an accurate GVT (or at least a good estimate of the GVT) that we can update frequently. To avoid idle time due to running out of work within the time window, we need to constantly recalculate the GVT *before* we've run out of work. This frequent recalculation of the GVT, during which the simulation is stopped, comes at a significant cost. To avoid it, we designed an asynchronous GVT estimation algorithm that we could call frequently and which would run in parallel with event processing. We implemented a multi-phase asynchronous GVT algorithm that relied on two phases of collection of event and cancellation send and receive data. This approach produced a very good lower bound on the GVT but had a high execution overhead. Our current approach is based on this scheme, but works in a single phase, using much less overhead and generating less network traffic than the original.

As the adaptive synchronization strategy developed, we decided on several qualities for the GVT algorithm:

- Low overhead: Use a simplified algorithm and sacrifice accuracy since speculation, especially in the final adaptive strategy, makes GVT accuracy less important. Of course, the GVT estimate must still be a lower bound on the actual GVT, thus accuracy here implies how close the estimate is to the actual GVT while remaining less than or equal to the actual GVT.

- Asynchronous: It is not practical to handle fine-grained simulations with algorithms that require all processors to stop the simulation.

- Balanced, centralized nature: some centralization is necessary for even a loose lower bound on the GVT estimate.

- Make GVT operations low priority: All GVT operations are performed on processors in idle periods when all events and cancellations have been exhausted.

We base our algorithm on distributed snapshot algorithms[35] which allow for asynchronous calculation of the GVT. Our approach closely resembles Mattern's GVT approximation algorithm[35] because it does not require FIFO message delivery order, which is not guaranteed in Charm++. We keep track of transient messages in simple local data structures that are later combined to determine what virtual times may be associated with transient messages[29]. Our concept of minimizing timestamps over intervals involves additional local calculation but simplifies GVT estimation considerably and minimizes network traffic.

7.2 The GVT Algorithm

We define the *object virtual time* (OVT) of an object o_i or OVT_i to be the point in simulated time to which the object has progressed. Each object starts out with an OVT of some initial value specified by the user (typically zero). Events may elapse time which increments the object's OVT. The OVT also advances when it executes the next event and that event has a timestamp greater than the object's OVT. In this case, the OVT advances to the event's timestamp. Rollbacks may also cause the OVT to be restored to an earlier virtual time.

Considering the set of all simulation objects O_p on a processor p, we then define *processor virtual time* (PVT), or PVT_p to be the minimum OVT of all objects in O_p. It follows that the GVT is the minimum PVT of all processors. Thus the following should hold:

$$\forall p \in \{p_1, ..., p_P\} \forall o_i \in O_p \ \ GVT \leq PVT_p \leq OVT_i$$

where P is the number of processors.

This definition works when we use a synchronous method that halts the entire simulation. Since determining the exact GVT at some point in time requires that there are no messages in transit (and thus an inefficient freeze on simulation progress), we settle for a good GVT estimate instead. Our GVT estimation algorithm operates by periodically requesting PVT estimates from all processors, as well as a table of event and cancellation sends and receives sorted by their associated timestamps. All information is gathered asynchronously and when combined, successfully produces a GVT estimate that is a lower bound on the actual GVT.

The GVT is triggered initially by a broadcast to all processors to send a *PVT estimate report* (discussed below) to an initial processor designated to compute the centralized part of the algorithm. When the centralized part receives all the reports, it calculates the GVT estimate and sends it to all the processors. Upon receipt of the new GVT estimate, each processor sends itself another message to compile the next PVT estimate report. The duty of computing the centralized portion of the algorithm is passed on to a next processor in a round-robin fashion. All messages exchanged by the components of the GVT algorithm have a lower priority than all event and cancellation messages, so GVT-related activities are only performed when processors run out of simulation work to do. Note that a processor that becomes idle and thus sends out its PVT estimate report does not need to stay idle. It may receive events or cancellations from other still active processors and can execute any incoming work received.

A processor's *PVT estimate report* consists of two items. First, the *PVT estimate* is the minimum *safe time* of all objects on that processor since the last time a PVT estimate report was sent. The safe time st_o of an object o is calculated as

$$st_o = min(max(ovt_o, wt_o), rb_o, ec_o) \tag{7.1}$$

```
startPhase()
  objs.Wake();  // wake objects to make sure all have reported safe times
  // compute PVT estimate
  PVT := POSE_UnsetTS;
  for (i=0; i<objs.numObjs(); i++)
    if ((PVT = POSE_UnsetTS) || (objs.objs[i].OVT() < PVT))
      PVT := objs.objs[i].OVT();
    end if
  end for

  um := SendsAndRecvs->PackTable(PVT);   // pack send/recv data and PVT into um
  send um via reduction to computeGVT on PE gvtTurn;
  gvtTurn := (gvtTurn + 1) mod NumPes;   // get next GVT location
  objs.SetIdle();                        // Reset safe times on objects
end
```

Figure 7.1: A processor prepares PVT estimate report

from the object's OVT, the timestamp of the earliest available event in the object's event queue(wt_o), the timestamp of the earliest pending straggler (rb_o) and the timestamp of the earliest pending cancellation message (ec_o). The safe time is the earliest timestamp of an event or cancellation that could be generated by an object in its current state *assuming no additional events or cancellations are received.*

This however does not take into account messages in transit. For this, we need to send a second item: a list of timestamps with corresponding send and receive counts. To obtain an accurate GVT estimate that is a lower bound on the actual GVT, we must keep track of the numbers of event and cancellation messages that were sent or received at each timestamp. These are stored in tables on each processor. Only relevant data is sent, so no send/receive data with timestamps later than the PVT estimate is transmitted. See Figure 7.1 for the PVT estimate report code.

All processors send their PVT estimate reports to the processor computing the centralized component when they run out of work to do. They may continue doing work if any exists after sending the report. The centralized GVT estimation algorithm determines the minimum of all PVT estimates as an initial GVT estimate. The send/receive information sent in the PVT

```
computeGVT(um)
  // see if message provides new min optGVT or conGVT
  if ((estGVT = POSE_UnsetTS) || (um->PVT < estGVT))
    estGVT := um->PVT;
  end if
  addSR(SRs, um->SRs, estGVT, um->numSRs);  // add send/recv info to SRs
  done++;

  if (done = NumPes) // all PVT reports are in
    // Check if send/recv activity provides lower possible estimate
    tmp := SRs;
    POSE_TimeType earliestMsg := POSE_UnsetTS;
    while (tmp && (tmp->timestamp <= estGVT) && (earliestMsg = POSE_UnsetTS))
      if (tmp->sends != tmp->recvs)
earliestMsg := tmp->timestamp;
      end if
      tmp := tmp->next;
    end while
    if ((earliestMsg < estGVT) && (earliestMsg != POSE_UnsetTS))
      estGVT := earliestMsg;
    end if

    broadcast setGVT(estGVT) to all processors;

    // cleanup
    estGVT := POSE_UnsetTS;
    earliestMsg := POSE_UnsetTS;
    free SRs;
    done := 0;
  end if
end
```

Figure 7.2: Calculate new GVT estimate

reports are consolidated into a single list sorted by timestamp. We then search through the sorted timestamps up to the current GVT estimate until a timestamp with non-matching numbers of sends and receives is found. If such a timestamp is found, the GVT is set to this timestamp. This algorithm is detailed in Figure 7.2.

This GVT estimate is then broadcast to all processors. Each processor then performs as much fossil collection as possible given the new GVT estimate. Each processor then awakens

```
setGVT(POSE_TimeType newGVTest)
  estGVT := newGVTest;
  objs.Commit();  // call fossil collection on all the local objects
  send message to startPhase() on self;
end
```

Figure 7.3: Processors receive new GVT estimate

any objects which may have had pending work that was too speculative to execute given the previous GVT estimate. Finally, each processor sends a low priority message to itself to prepare the next PVT estimate report as shown in Figure 7.3.

Proof of Correctness

Our GVT algorithm is correct if it produces a GVT estimate g which is a lower bound on the actual GVT. Thus we want to show that g is a lower bound and that there is no earlier time h at which a new event or cancellation can occur. If such an h exists, it might mean we had committed events during fossil collection that are now needed to recover state in a rollback.

Definitions: Suppose there are M_i objects on processor i. An object o_{im} has a safe time st_{im} determined as in Equation 7.1. Let p_i be the PVT estimate on processor i such that $p_i = min_{m \in \{0,\ldots,M_i-1\}}(st_{im})$. Let $p = min_{i \in \{0,\ldots,P-1\}}(p_i)$, the minimum of all the PVT estimates. Let L be a list of 3-tuples of the form (t, s, r) such that L is sorted on the first value t which is the timestamp of a number of sends s and receives r. Let k be the tuple (t_k, s_k, r_k) such that t_k is the earliest timestamp in L such that $s_k \neq r_k$. L represents the merged list of send and receive information obtained from all processors. Let e be t_k. According to the algorithm, $g = min(p, e)$. Finally, we define a *PVT interval* on a single processor as the wall clock time between when the processor sends one PVT estimate report and the next. Over an entire parallel simulation, a *GVT interval* is not defined by time but rather is the set of intervals, one on each processor, corresponding to a particular GVT estimate calculation.

This interval structure is pictured in Figure 7.4. Each different PVT interval has a different pattern in the figure, while the similarly patterned PVT intervals together make a single GVT interval. Note that the *computeGVT* algorithm gathers all PVT estimate reports and computes a GVT estimate for the GVT interval prior to the one in which it is located. PVT estimate reports are sent via a reduction and not directly to *computeGVT* as the figure shows.

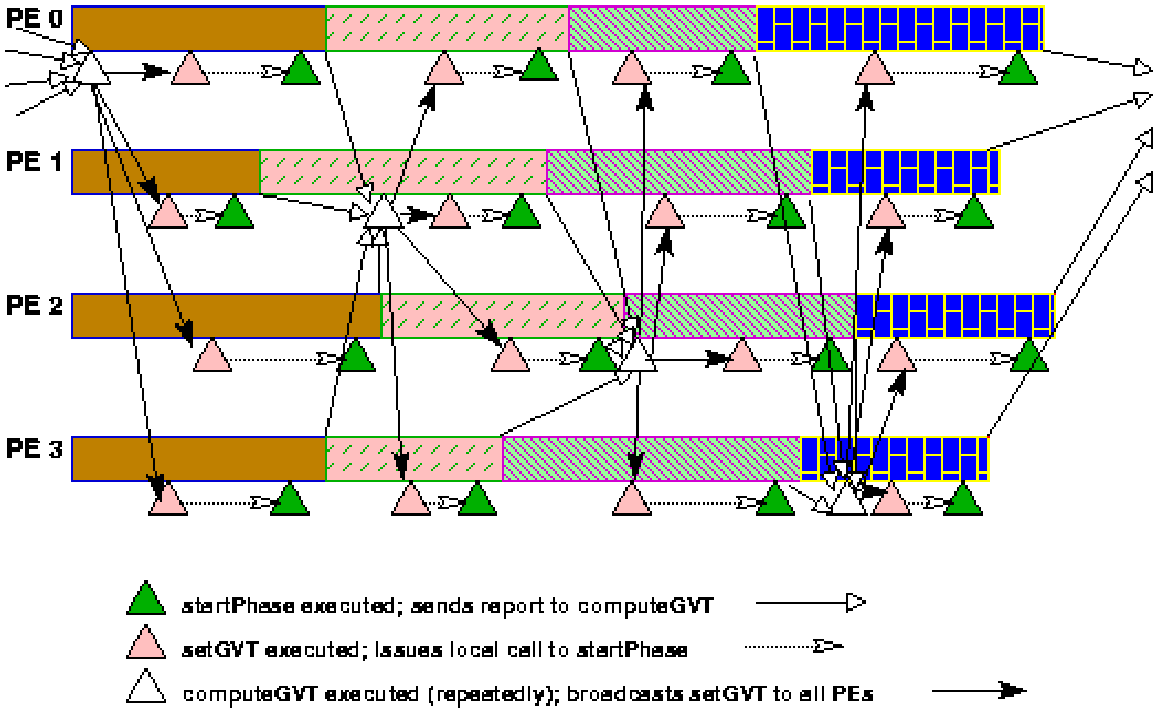

PE 0
PE 1
PE 2
PE 3
startPhase executed; sends report to computeGVT
setGVT executed; issues local call to startPhase
computeGVT executed (repeatedly); broadcasts setGVT to all PEs

Figure 7.4: PVT and GVT intervals

Theorem: Given a GVT estimate g calculated by this algorithm, no events or cancellations with timestamp $h < g$ can be generated and no outstanding sends or receives with timestamp h exist.

Proof: We proceed with a proof by contradiction and assume that we have some timestamp h such that $h < g$ at which something occurs after we have computed the GVT estimate g. Suppose that g is computed in GVT interval $i + 1$ for GVT interval i. Let us first consider the case where $i = 0$. Since every object starts at time zero with an OVT of zero, all safe times for all objects for all processors must be zero, so the first g is always zero. There can be no $h < 0$.

Now we consider an arbitrary interval i. What can happen at time h in interval $i+1$? We could receive an event or cancellation with a timestamp h that could cause the generation of other cancellations or events. Over the course of PVT interval i on each processor, no object ever had a safe time less than p, the minimum of all PVT estimates, by definition. So for g to be incorrect, we must discover something in interval $i+1$ with timestamp $h < g$. Since the definition of safe time applies to the state of an object assuming no additional events or cancellations are received, this must be the means by which we could obtain a lower timestamp than g.

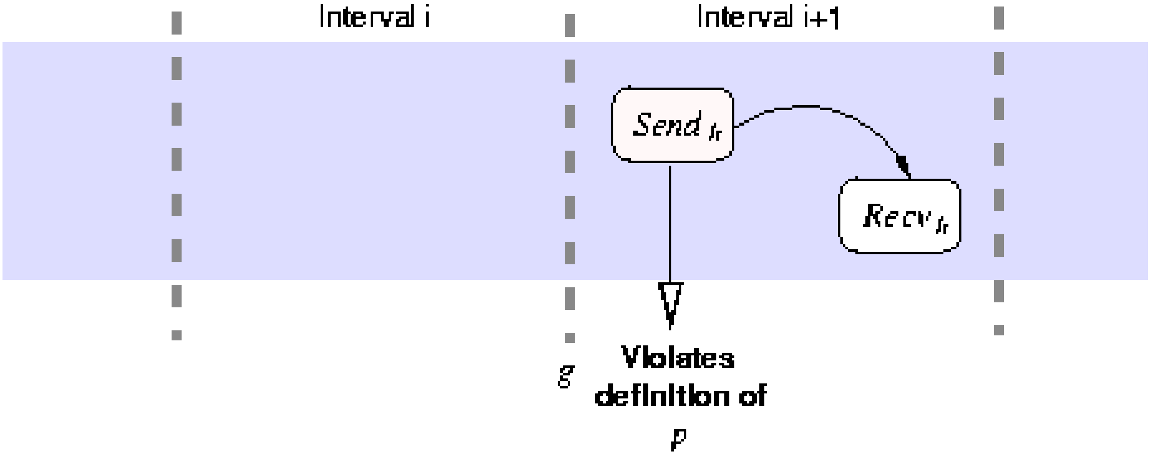


Figure 7.5: A receive in interval $i+1$ earlier than g cannot ultimately be caused by an event in the same interval

Thus we suppose that we have received an event or cancellation with timestamp h in interval $i + 1$ as shown in Figure 7.5. We must determine where this event came from. It must have originated on an object with a safe time less than or equal to h (we will assume equal to h for simplicity). By definition of p, this object could not have sent the message while in interval i as shown in Figure 7.6. If the object sent the message from interval $i+1$ as in Figure 7.5, and we know the object previously had a safe time greater than or equal to p, the object must have rolled back in interval $i+1$. Yet another message would be required to cause *this* rollback. Since events do not materialize spontaneously, causality requires that there must be an point of origin for the message of timestamp h outside of interval $i+1$. We have eliminated i from the possibilities and must consider intervals prior to i.

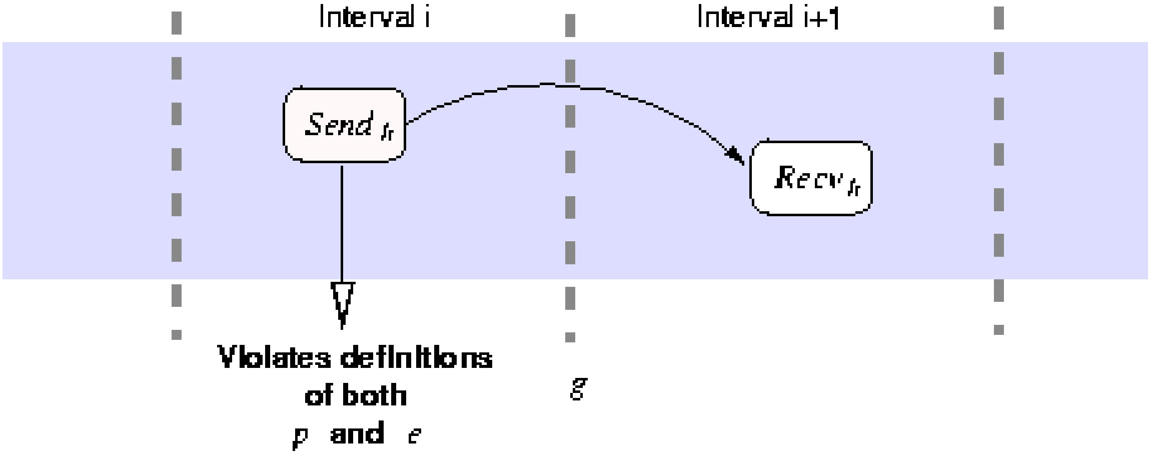


Figure 7.6: A send from the previous interval requires that the sender have a safe time less than g and also would be accounted for in e as an unanswered send at time h

If a message with timestamp h was sent from an interval prior to i but was not received until interval $i+1$, then the e calculated from interval i must by definition be h since a receive at this time is unaccounted for. Thus since e is not h, there can be no sends in intervals prior to i that did not have corresponding receives until interval $i+1$. This is shown in Figure 7.7. Thus no such message with timestamp h can be received in interval $i+1$. □

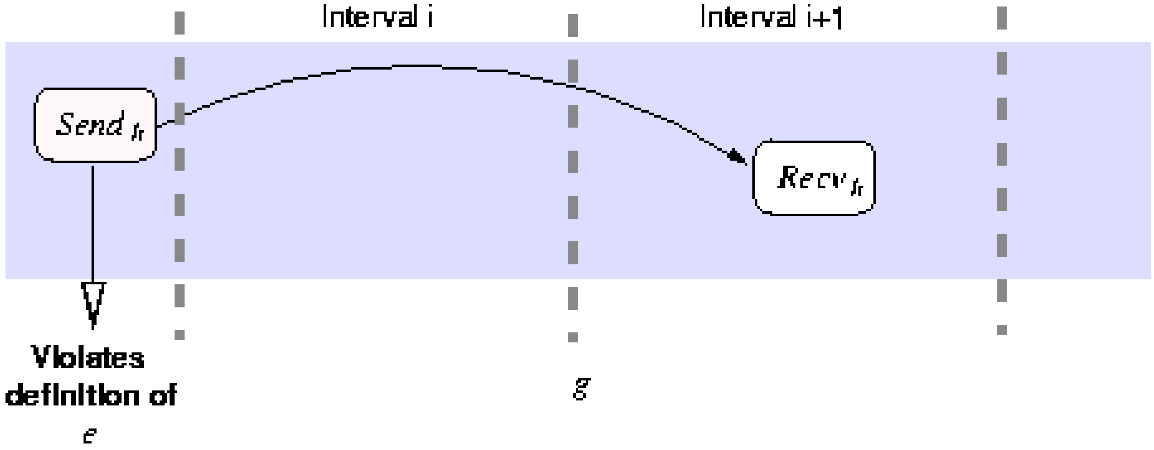


Figure 7.7: A send from some earlier interval prior to i should be accounted for in e

7.3 GVT Scalability

As mentioned earlier, our GVT estimation algorithm has a centralized component which rotates from processor to processor in a round-robin fashion, so that no one processor gets too far behind the rest due to GVT computation. Since the algorithm does not require the storage of any data at the central calculation point that is reused in the next iteration, making this component mobile helps to avoid the usual drawbacks of a centralized strategy. It thus provides a rudimentary means of load balancing the overhead required for GVT calculation.

In Chapter 4, we discussed the need for PDES support algorithms that can scale superlinearly. We showed how many of the overhead sources scale in Pose in Figure 5.3 of Section 5.6 where we discussed the scalability of overhead. Our GVT algorithm is one overhead component which scales quite well. We ran our synthetic benchmark on Tungsten with a moderately fine grainsize (6 μs on average) and over 2 million events up to 64 processors and obtained the speedup results shown in Figure 7.8. This graph also plots average (per processor) GVT speedup which demonstrates superlinearity and does not degrade when the overall speedup does. We also plot the speedup of the maximum time spent on GVT on a processor. This differs only slightly from the average case and shows that the GVT computation is well balanced over the processors.

There is a slight dip in GVT speedup at around 16 processors. At this point the amount of simulation work to be done on each processor has dropped considerably and thus the GVT runs much more often. Each iteration however has much less work to do, so the performance does not degrade significantly. In fact, this behavior is the primary reason for the superlinear speedup of GVT overhead. The decentralized component of the GVT algorithm determines the minimum OVT on each processor and collects and summarizes send and receive information for the centralized component. The minimum OVT calculation does not change over each iteration and involves only a little computation. The expensive part of the calculation

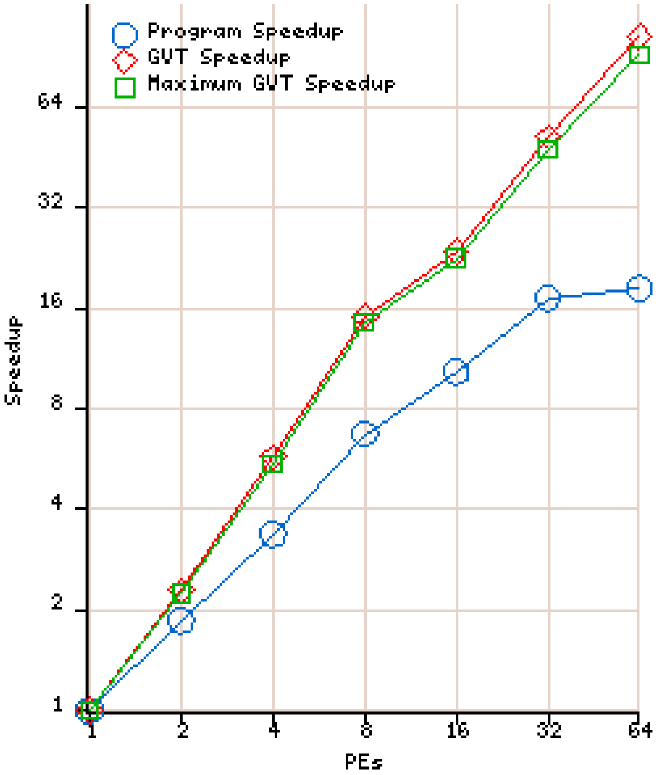


Figure 7.8: Scalability of GVT estimation

is the collection and manipulation of the send and receive data. The data structures involve small amounts of sorting and searching. Since our synchronization strategies impose limits on speculation to avoid rollback, they are likely to invoke the GVT more frequently. Thus the operations on GVT data structures are designed to have good average case performance as the associated data structures grow large and best case performance for smaller data structures[1] We can see this effect when comparing the GVT behavior of the OPT strategy to ADAPT2 as shown in Table 7.1. OPT invokes the GVT rarely, while the limits to speculation in ADAPT2 result in more frequent GVT iterations. The table shows the parallel time (average per processor) spent on GVT computation (which is why it scales) as well as

[1] The data structures in question exploit the fact that send/receive information is likely to arrive in timestamp order. The larger the data grows and the longer the period of time over which we are adding to the data structure, the more likely we are to violate that timestamp ordering. Thus constant time operations become search operations.

the number of iterations performed throughout the simulation and the average sequential time per iteration.

Strategy	Time	Iterations	Time/Iter.
2 Processors			
OPT	22.51	3	15.01
ADAPT2	21.09	43	0.981
8 Processors			
OPT	5.34	4	10.69
ADAPT2	2.45	43	0.456
32 Processors			
OPT	1.18	4	9.428
ADAPT2	0.42	44	0.303
128 Processors			
OPT	0.27	4	8.749
ADAPT2	0.10	44	0.304

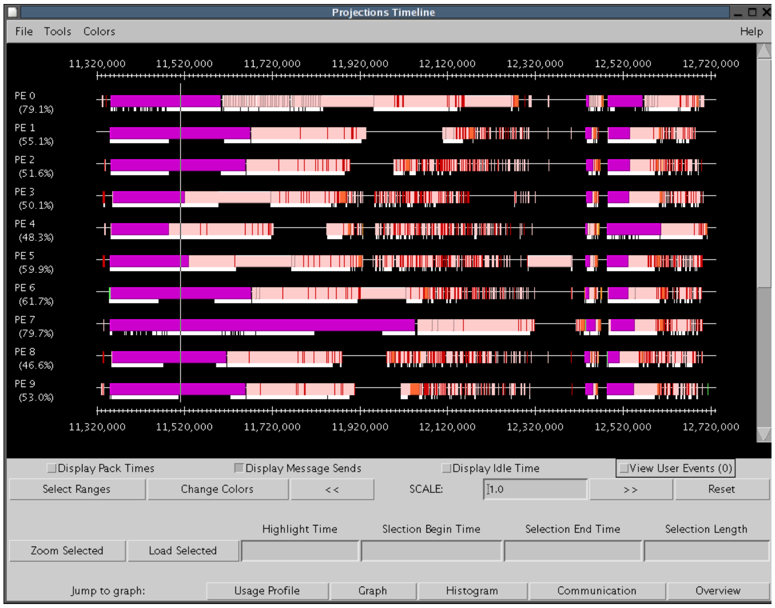
Table 7.1: GVT takes less time when run more frequently

When we add processors, the size of the data structures on each processor decreases. The superlinearity arises from the fact that we exploit the probable timestamp ordering of send and receive records added to the data structure to achieve best case performance for search and sorting operations. Thus per iteration GVT cost decreases. In addition, as we increase the number of processors, the work per processor goes down, processors go idle more often, and the GVT is invoked more frequently. This results in even smaller data structures as they are more frequently purged of older information. Thus the time per GVT iteration decreases even further as the number of iterations increase, but since each operation is even more likely to be best case performance, total time on GVT reduces even more dramatically.

7.4 GVT in Action

In the previous chapter, we discussed a very small benchmark run with reference to Figure 6.16. Below, we show a timeline for about $\frac{1}{6}^{\text{th}}$ of the program near the end. This is a much more detailed view than the overviews shown in the previous chapter. Again, time is on the

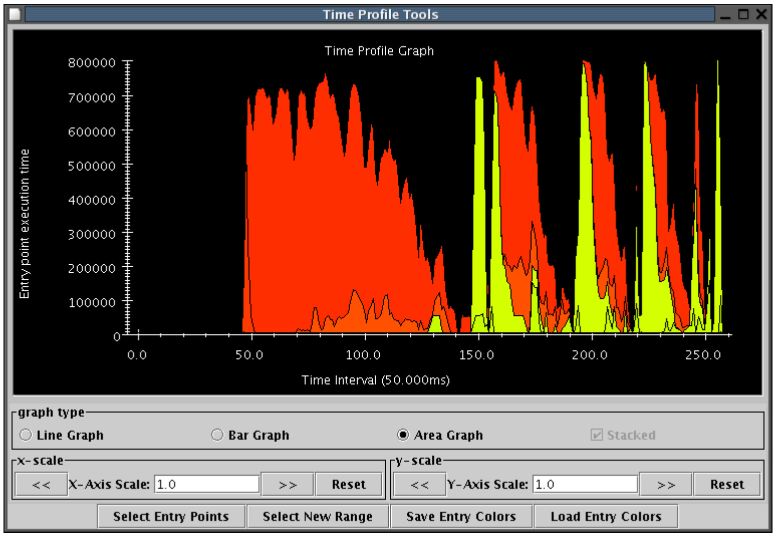
x-axis and the processors are on the y-axis. Entry points are plotted by color. This shows three GVT intervals starting at the beginning with a new GVT estimate received on all the processors. This dark (purple) portion represents the the *setGVT* entry method on each processor which checks each local poser to see if any events were enabled by the new GVT estimate and if so, executes them. Following this in the paler shade (pink) are new events received and executed on the processor. As each processor runs out of work, the *startPhase* entry method is called to prepare the PVT estimate report.



SCALE: 1.0

Figure 7.9: Timeline showing *setGVT* and event handling on processors

Another view of the entire program is presented in Figure 7.10. This graph compares the work performed by each component over the duration of the program. The tall pale (yellow) peaks represent the *setGVT* method and the short pale peaks are *startPhase* operations. *computeGVT* is so fast it doesn't even show up here. Given that most of *setGVT* is spent executing simulation events, this chart shows that the time spent on calculating GVT is not significant for this run.



graph type

○ Line Graph ○ Bar Graph ● Area Graph ☑ Stacked

x-scale

X-Axis Scale: 1.0

y-scale

Y-Axis Scale: 1.0

Figure 7.10: Time profile showing time spent on GVT vs. simulation

Chapter 8

Communication Optimization

Any parallel/distributed system can benefit from a well designed communication framework that optimizes a variety of communication patterns. What further factors should be considered when pondering communication in PDES? For one, communication is likely to be irregular and varied such that standard collective communication optimizations do not suffice. How can we use or modify existing communication frameworks to reduce the overall cost of communication in PDES? We consider these questions in this chapter and present new strategies for optimizing communication.

What do we need to communicate in PDES? The types of messages we send can be roughly categorized as *user messages* and *system messages*, based on the POSE user's awareness of the messages. The only type of message that fits the user category is the event message. System messages are: *cancellation messages, GVT gather/broadcast messages, load balancer gather/broadcast messages* and *object migrations*.

Of these, the event and cancellation messages are prioritized – sorted into the scheduler when received, while the rest are considered lower priority. What if we could develop a communication framework that could make use of the priority attached to these messages *before* sending them? We may be able to hold low priority events and package them into a single send operation. We might also want to use our library to expedite the delivery of certain high priority messages, such as cancellations and low timestamp events. We could make use of the speculative window to deliver events within that window quickly, and to save later events for later delivery.

Another aspect to communication is the *pattern*. Many applications benefit from highly structured collective communication optimizations. General-purpose tools such as Pose however, need to support a wide variety of irregular problem structures. In particular, fine-grained PDES is notoriously challenging for its small amounts of computation coupled with large amounts of communication. In addition, the triggering of events does not typically involve the transfer of huge quantities of data. Thus, the problem we are facing is one of optimizing the overhead of sending plentiful small messages.

8.1 Streaming Communication Optimization

Charm++ has several existing collective communication optimization strategies developed by Sameer Kumar and others at the Parallel Programming Laboratory [21]. However, application developers must currently select a strategy and choose parameters that work well for the specific application they are working with. These strategies work with the Charm++ delegation framework and are particularly tuned to applications with distinct periods of time during which massive quantities of data are exchanged. Because we are striving for a general purpose simulation environment, we want to handle irregular applications that may not fit this or any other known communication pattern. Thus we have developed communication strategies for Pose to handle constant communications.

A straightforward streaming communication strategy was developed in Charm++ to reduce communication overhead by gathering together many small messages into a single message send. In this approach, each processor has a set of outgoing message buffers, one for each other processor. Messages are stored in these buffers according to their destination processor. The buffers are eventually emptied by packing all the messages together and performing a single send operation per outgoing message buffer. The strategy is parameterized by a buffer size and a timeout period. Thus the packing and sending process happens when either the buffer is full or the timeout period has passed.

Consider a simulation in which k messages are transmitted to remote processors. Using a simple communication model, the time to transmit a single point-to-point message is roughly

$$T_{ptp} = \alpha + m(\beta + \gamma)$$

where α is a per transmission cost for sending, receiving and network overhead, m is the size of the message in bytes, β is the per byte transfer cost and γ is the per byte copy cost (for packing messages into a single buffer). Our strategies trade off a potential delay in the arrival of messages for a reduction in the α cost of message transmission. For example, communication overhead for k messages is roughly $k\alpha + km\beta$, but if we manage to pack just two messages together on average, this cost is reduced to $\frac{k\alpha}{2} + km\beta$.

8.2 Mesh-based Streaming Communication Optimization

The mesh-based streaming communication optimization is similar to streaming in that messages are buffered, packed together and sent as a single message with a single send operation whenever the buffer is full or a timeout is reached. The difference is that the strategy considers the processors to be organized as a 2-D square mesh of P processors. Instead of each processor have P buffers for each possible destination, it has $\sqrt{P}$ buffers. A message destined for processor p is placed in the buffer of the processor q that represents the *row* of the mesh in which p is located. When the message is received on q, it is buffered again (assuming $p \neq q$) and eventually sent along the row to the appropriate column that is p.

In this scheme, most messages will need to make two hops before reaching their destination. However, if we can pack enough messages together, there can be a significant payoff. Consider the case where we manage to combine four messages together on average into one send operation. The communication overhead to send k messages is $\frac{k\alpha}{2} + 2km(\beta + \gamma)$. Thus we have halved the α cost while doubling the β cost. The more messages we pack together, the more the α cost is reduced while the β cost stays the same.

8.3 Prioritized Communication Optimization

We have extended the functionality of the streaming strategy to handle prioritized messages in an intelligent fashion. This approach operates identically to the streaming strategy except that a third case exists for when to pack and send the buffered messages. When a high priority message is added to a buffer, the entire buffer is packed and sent to the destination processor.

What is a high priority message? This is dependent on the application. The prioritized streaming strategy lets the user specify a high priority h. Any messages with a priority equal to or higher than h triggers an immediate packing and sending of the target buffer with the high priority message in it. Messages with lower priority are buffered. The same overflow and timeout parameters also apply.

This however is not sufficient as a strategy for prioritized streaming in PDES. Messages are prioritized by timestamp, and the higher the timestamp, the lower the priority. However, the minimum timestamp (the highest priority) constantly increases throughout the duration of the simulation. Thus we need a way to update h with the prioritized streaming strategy. In addition, the streaming strategy must keep track of what is the highest priority of the messages in a buffer. When h is updated, it examines each buffer's highest priority and packs and sends those which have high priority as defined by the new value of h. POSE's GVT estimation code updates h whenever a higher GVT estimate is calculated.

Thus, the prioritized streaming strategy guarantees that no early timestamps languish in communication buffers by expediting their delivery.

8.4 Performance of Communication Optimizations

We ran our large and small instances of the synthetic benchmark from Chapter 6 on Tungsten with ADAPT3 and four separate communication optimizations: streaming, mesh streaming, prioritized streaming with high priority set at GVT (Prio1) and prioritized streaming with high priority set above the GVT by an average speculative window size (Prio10). We configured the streaming optimization strategies to flush their buffered messages when the buffer size reaches 20 or a 2 ms timeout is passed. The results are shown in Figure 8.1.

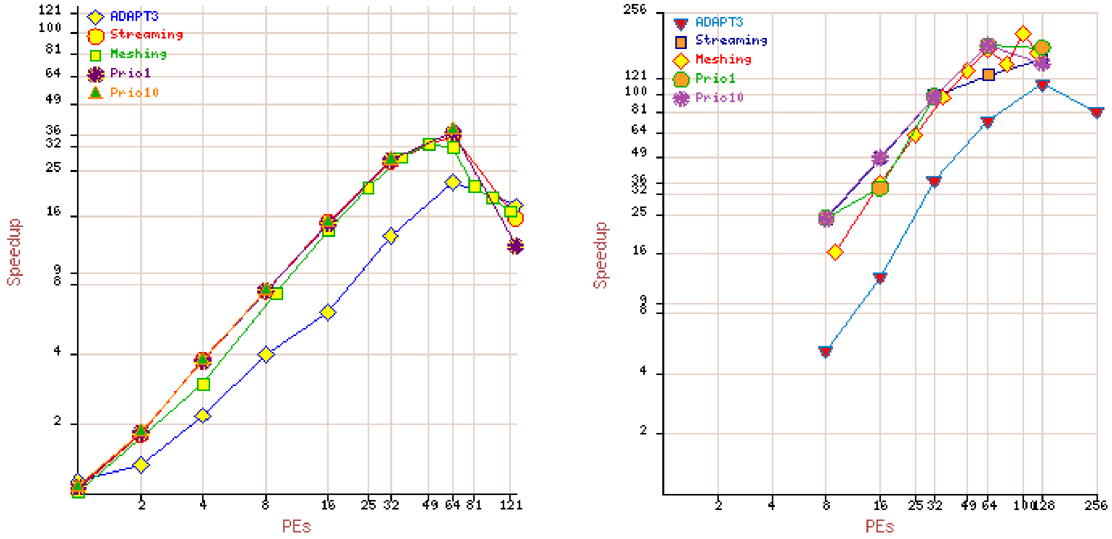


Figure 8.1: ADAPT3 with various communication optimizations on small and large problem instances

All communication optimizations improved the performance of ADAPT3 considerably. However, beyond (and sometimes before) ADAPT3's peak performance, using a communication library produced some erratic behaviors in Pose. It is important to note that any kind of mechanism we use that involves holding back messages defeats the purpose of adaptive speculation. Without future events present, an object cannot take advantage of executing them in bunches, nor can it get an accurate estimate of its future behavior. Thus a trade-off exists between communication optimization and synchronization performance. Also, holding back messages might also cause idle time on a processor which in turn may cause the GVT estimation to execute more often. These factors contributed to an occasional anomalous run that lasted an extremely long time. Of the strategies shown, Prio1 exhibited to most stable behavior and achieve some of the best speedups for the simulation.

8.5 Summary and Future Research

In this chapter, we have shown the benefits of using communication optimizations for constantly streaming messages in PDES. The possibilities for future explorations are manifold. In particular, we suggest the investigation of more adaptive behaviors for communication optimizations.

Specifying the minimum high priority h provides us with some control over high-priority message delivery, but possibly not quite in the way we would wish. For example, we may want certain messages to arrive quickly, but not all the others that are flushed when the high priority message is added to a buffer.

One approach to this would be an adaptive communication strategy that is built in to an adaptive synchronization strategy such as ADAPT3. This communication strategy would use the prioritized streaming strategy previously described to handle messages that have timestamps in the distant future. Messages in the immediate future should be sent directly. Messages in between should be sent one way or the other based on predictions of how far in the future they will be on the objects they arrive on. This is difficult since an object only knows about its own behavior and not about the behavior of other objects. Its own behavior could be the predicter: if it has very little speculative work to do, it should send events immediately. If it has plenty of events in its queue, it would delegate the message transmission to the prioritized streaming strategy.

Chapter 9

Load Balancing

Parallel discrete event simulation presents us with a unique opportunity to study different styles of load balancing in a new environment. In addition to improving processor utilization, we can also use load balancing to improve the rate of progress in a simulation and as a result improve the simulation's performance.

A unique side effect of optimistic synchronization and speculation is that the most common measure of load (or lack thereof), idle time, may not be present. In short, having all processors busy is not an indication of perfect balance since there may always be some speculative work to do. Thus, some of the busy processors may be busy speculating, rolling back and re-executing. This requires us to redefine not only the concept of *load*, but also what it means to *balance* it[41].

9.1 Dynamic Load Balancing in Optimistically Synchronized PDES

We would like a simulation to adjust to changes: changes in the parameters, changes in the simulation behavior over time, changes in the computation power available during the course of a run. To obtain better speedups, we need to make sure every processor has maximally useful work to do, and since simulations can involve highly irregular computations and strong data dependence, we need to periodically adjust the distribution of work via some sort of load balancer.

We could use a standard Charm++ approach to load balancing[5], by monitoring computation load and communication[28], but this would be inadequate for the situations that arise from optimistic synchronization. For example, a processor may have a set of objects that have ample quantities of speculative work to do, while another processor is executing non-speculative events at or very close to the GVT estimate. The former processor may appear to be heavily loaded from a computation standpoint, so we may be tempted to offload work to the latter. In reality, the best form of balancing would involve an exchange of objects between the two, so that both had some immediately useful work to do, and when that was exhausted, they could both speculate for a while until a better GVT estimate is produced.

Thus we need to experiment with more appropriate definitions of the concept of *load*. Clearly, we wish to use a form of prioritized load balancing that gives precedence to objects on the wavefront of simulation progress, the objects doing the least speculative work. We next describe the various qualities that comprise the load in a parallel discrete event simulation.

Object and Processor Virtual Times: Earlier, we defined *object virtual time* (OVT) on an object to be the point in virtual time to which the object has progressed, and *processor virtual time* (PVT) is the minimum OVT on a processor. Thus the following should hold for all processors i and objects o_i : $\mathrm{GVT}_est \leq \mathrm{PVT}_i \leq \mathrm{OVT}_{o_i}$. Since objects with OVTs closer to the GVT estimate are the least likely to be rolled back and the most likely to be performing non-speculative work, we want to give these objects precedence, so we consider these objects to represent a *heavier* load than the others. Similarly, a processor with a PVT close to the GVT estimate is likely to have many such objects, and is thus considered to have a heavier load.

Communication: Like more standard load balancers, we should take communication into consideration. We wish to minimize the quantity and size of events that must cross processor boundaries.

Idle Time: In problems with a low degree of parallelism, we may indeed encounter idle time on processors. We may also encounter it even in highly parallel problems, when the work on one or more processors dwindles and explodes on others. In the latter case, there may be plenty of work to spread around, and a single call to a load balancer could fix the problem for some time. However, in the former case, we might need to continuously move work around to keep all processors busy.

Rollbacks: If an object has been rolling back a large percentage of its events, we want to give it less opportunity to do speculative work, so we weight the object as lighter than objects that rarely get rolled back. This makes it possible for "heavier" objects to be moved to processors with lighter objects, so that they can make better use of the CPU time.

Past and Future Events: An object keeps track of recently executed events and may also have some future events locally queued. We can use the frequency of the past events and the presence of the future events to gauge both the quality (in terms of speculativeness) and quantity of its future workload. An object with a huge amount of work on the horizon has a heavier load than an object with no future work available. Similarly, if an object has no future work, we can guess that it will probably receive some soon if it was very busy in the immediate past, and weight it to be heavier than an object that has been completely idle.

Object Size: The quantity of state data encapsulated by the object is worth considering separately since we want to avoid migrating extremely cumbersome objects. Both the size and the complexity of the data structures used can heavily impact the time it takes to pack up and migrate an object.

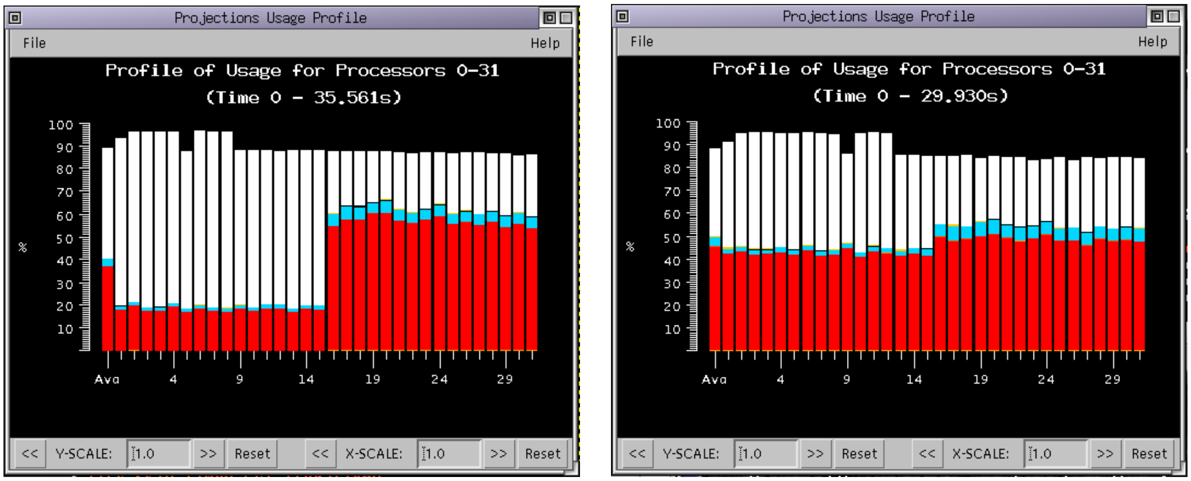
9.1.1 A Basic Load Balancing Framework for POSE

We have built a fully asynchronous load balancing framework into POSE that operates concurrently with the ongoing simulation. This is a periodically-invoked, balanced centralized strategy which gathers local load information about objects from each processor, determines how much work each processor should try to relocate, and sends that information back to the processors which then attempt to move work around to improve locality of communication without breaking existing local tight-communication ties. Within this framework, we have implemented a simple strategy which uses the OVT and communication criteria from above.

Communication data concerning each event send and receive is already being collected for use by the GVT estimation algorithm. We reuse this communication table in our load balancing framework. When the load balancer is invoked, a chare group object (one per processor) called TheLBG gathers information about the local objects according to the load balancing strategy being used. In the case of this example, it applies the OVT criteria to weight each object depending on how close to the current GVT estimate the OVT falls. If the object is idle (has no known future work to do), it has the lowest weight. The object weights are then summed and reported to the central LBstrategy object (which changes to a different processor each time, just like the centralized GVT component). This object sorts the processors by load, and then creates a mapping which will instruct each overloaded processor to migrate a certain amount of work to a set of underloaded processors. This information is broadcast back to each branch of TheLBG. The local branches use the data to determine if they are overloaded, and if so, who to send work to. They then start to migrate simulation objects away selecting objects based on information in the communication table.

We have tested this simple load balancing strategy on a simulation with an initial object imbalance, fine-grained events, and low-to-moderate degree of parallelism. Because the processor usage is not high, we can show how load balancing spreads the work out better across the available processors. This represents the more traditional load balancing problem of trying to maximize the utilization of processors. Figure 9.1(a) shows the processor utilization of the program run without the load balancer and Figure 9.1(b) shows the same program run with the load balancer on.

As we see in the second usage graph, the average percentage utilization has improved from



(a) (b)

Figure 9.1: (a)Imbalanced utilization in a problem with object imbalance; (b) Smoother utilization with load balancing

40% to 50% with the addition of load balancing, and idle time has been reduced. We also experienced an improvement in overall runtime from 35.6 to 29.9 seconds, but the overhead of the load balancer was a significant component of the runtime for this small problem.

9.2 Speculation-based Load Balancing

Our simple load balancing strategy uses the OVT of an object as a measure of load balance. However, OVT is mostly an indicator of an object's progress through a simulation and not necessarily of how much work in terms of computational load the object has performed. Thus we need to obtain more information about the object's past event behavior to estimate load more accurately.

An object's speculative behavior is indicative of how great a contributor that object is to the load of a simulation in terms of useful simulation work and rollback overhead. In speculation-based load balancing we make use of an object's percentage of useful work executed (see Chapter 6) in addition to its OVT to determine its load. This is particularly important because in the simple OVT-based strategy, we noticed that the load imbalance leads to excessive speculation on processors with low utilization, and that this effect increases with the number of processors.

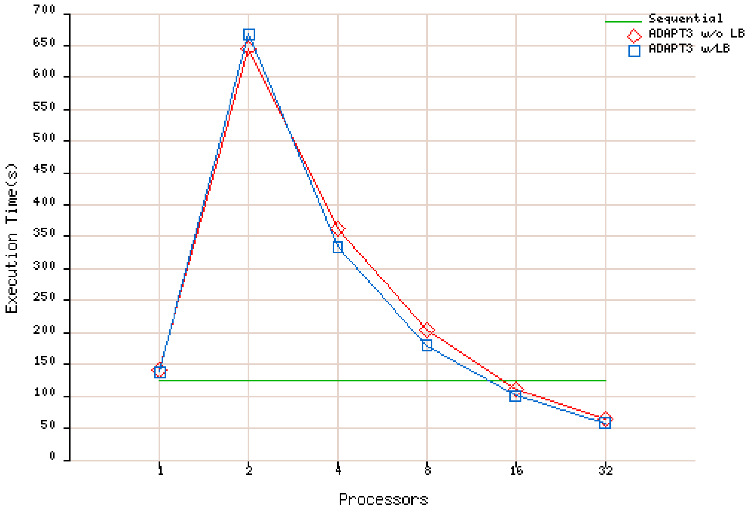


Figure 9.2: Execution time for ADAPT3 with and without load balancing

We ran our benchmark with a larger imbalanced problem instance, with posers on half the processors representing 25% of the load, and the rest representing 75% of the load for a total of 5000 posers. We ran the experiment on the Cool cluster. While load balancing did improve the performance considerably, we were aiming for a significant reduction in the speculative load, which we were not yet able to achieve. In fact, as Table 9.1 shows, speculation displayed a slight *increase* when we used load balancing, and on the whole, the percentage of useful events executed remained low. Clearly, this approach will benefit from further study.

We show the execution times for the experiment in Figure 9.2. On two processors we take a huge performance hit, and load balancing is unable to recover from it, but beyond that, load balancing always improves performance. Note that speedup relative to sequential time is 2.19 at 32 processors with the load balancer, versus 1.97 without, and 1.24 on 16 with load balancing versus 1.14 without. The most significant improvement occurred at 8 processors where load balancing reduced the execution time by about 25 seconds.

A couple positive points to make about this load balancing strategy: 1) Computational

	W/o LB			W/LB		
PEs	Total Events	Useful Events	% Useful	Total Events	Useful Events	% Useful
1	223383	223383	100%	223383	223383	100%
2	579729	197355	34%	568987	197355	35%
4	495750	217836	44%	526851	217836	41%
8	536693	223486	42%	567124	223486	39%
16	540931	223772	41%	579317	223776	39%
32	512447	222834	43%	598879	222834	37%

Table 9.1: Speculation in an imbalanced simulation with and without load balancing

overhead due to load balancing was negligible (<240ms per processor at its maximum on 2 processors); 2) Additional communication overhead did not overwhelm the improvements (27 seconds of communication and system overhead maximum on 8 processors with load balancing versus 9 seconds without) even though no communication optimizations were used. However, we could certainly benefit from using one of the communication optimizations to migrate objects. These points suggest that we have successfully achieved the goal of a lightweight and effective load balancing framework for PDES. This lays an excellent groundwork for future work in PDES-specific load balancing.

9.3 Summary and Future Work

We have a load balancing framework in place in Pose in which a variety of load balancing strategies can be implemented. A basic load balancing strategy handles object imbalance and idle time very well using a combination of object OVTs to determine load and communication to determine placement. We have also implemented a strategy that uses the speculative behavior of an object as an estimate of load. This strategy needs further development.

A future direction of Pose load balancing is prioritized load balancing strategies. Some of the criteria we've discussed (such as OVT) have something in common: they estimate more of a *priority* of an object than an actual computational load. Consequently, we are translating high priority into high load. Ideally, we would like to balance the load so that each processor has an equal share of the high priority work as well as of all other priorities. This is because the passage of time will result in medium priority work becoming high priority, and low priority becoming medium priority, etc. Thus a prioritized load balancing strategy is another possibility for PDES. Another improvement would be in the usage of communication optimizations to bundle migrating objects together to reduce the communication overhead.

Chapter 10

Visualization and Analysis

Because of the challenges presented by optimistic PDES, this thesis has been devoted to improving and analyzing the performance of our parallel simulation framework. But the purpose of the simulation in the first place is to analyze a model of some real-world system. In this chapter, we discuss the utility of POSE as an analysis tool.

The simple methods by which one can extract information from a POSE simulation are as follows:

- `CommitPrint` allows the user to print information from events as the simulation is running. The print will not happen exactly when the event executes it, but instead when the event is committed. Events that are rolled back get there output deleted and if they are re-executed can generate new output. Whatever the final output was for the event gets printed when the event is committed. The user could choose to print information about the current state, the current virtual time on the object (OVT) or other statistics.

- Commit events allow the user to execute an code they like when an event is committed. This could be used for statistics collection and periodic reporting.

- When the simulation comes to a halt, there is a special terminus function for all objects. The user can write whatever code they wish to in this function. It is useful to examine the final object state in this function.

- At the beginning, the user can specify a callback function which Pose will hand over control to at the very end of the simulation. The user can execute whatever they like here, to finalize their analysis of the simulation.

Besides these straightforward methods for information gathering, Pose offers several other ways to improve both the model and the simulation performance of the model. These approaches are detailed in the following sections.

10.1 Visualizing Degree of Parallelism

In Section 4.3.2, we discussed the notion of *degree of parallelism* (DOP). We would like to know what is the maximum number of processors we can run a simulation on and still have a high degree of parallelism. We would also like to know how that might vary over the course of a simulation. In addition, DOP can be mapped onto virtual time instead of real time, and describe how efficient our *model* of some system (which presumably has simultaneously occurring activities) is. This is especially useful when trying to improve a model for some complicated process in which we are trying to maximize throughput while minimizing costly resources. Thus we have developed a means to view a graph of object activity over time — real and virtual — that accurately portrays the parallelism available throughout the course of a simulation, for both the model and the simulation implementation.

We use a version of the detailed contention-based network model simulation (discussed in the next chapter) to illustrate the use of these two analyses.

10.1.1 Degree of Parallelism of the Model

Our network simulation has a mode in which traffic patterns can be generated to flood the network with messages which are then packetized and transmitted throughout the network. In this particular example, traffic is generated with a Poisson distribution and transmitted to destinations with a uniform distribution. Each node transmits some 200 messages and the simulation runs until there is no work left in the system. We have simulated a 128-node network configured as the BlueGene/L torus network model. The degree of parallelism is graphed in Figure 10.1.

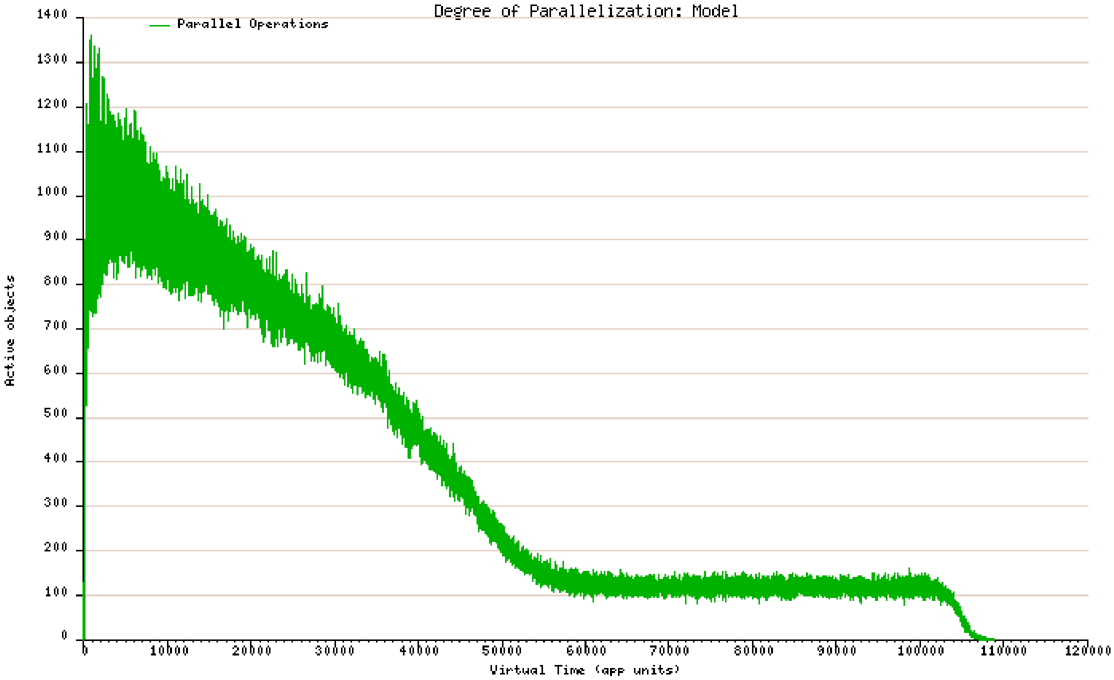


Figure 10.1: Degree of Parallelism of a 128-node interconnection network model

The graph simply shows how many objects (along the y-axis) are active at a virtual time unit (along the x-axis) over the entire virtual time that the simulation runs. In this case, we clearly see an expected pattern of traffic gradually working its way through the system, until there is no more work to do. The data for the graph is straightforward to generate: we log the OVT at the start and end of each event, then map the virtual time units onto bins, and for each event, increment a bin counter for the virtual times during over which the event elapsed time. Since only one event can be active on an object at a time, each overlapping event counts as an active object at that time.

10.1.2 Degree of Parallelism of the Simulation Implementation

Analyzing a DOP graph for the simulation implementation can give us an idea of how well we have translated a model into a simulation. It can also reveal potential bugs in the model or inefficiencies in the simulation. What we actually see is a graph of what the utilization would look like if we had infinite processors to execute the simulation on, with no overhead or communication time between events. It shows us the maximum possible overlap of work throughout the entire simulation. In Figure 10.2, we show the graph of the simulation DOP for the same problem as above.

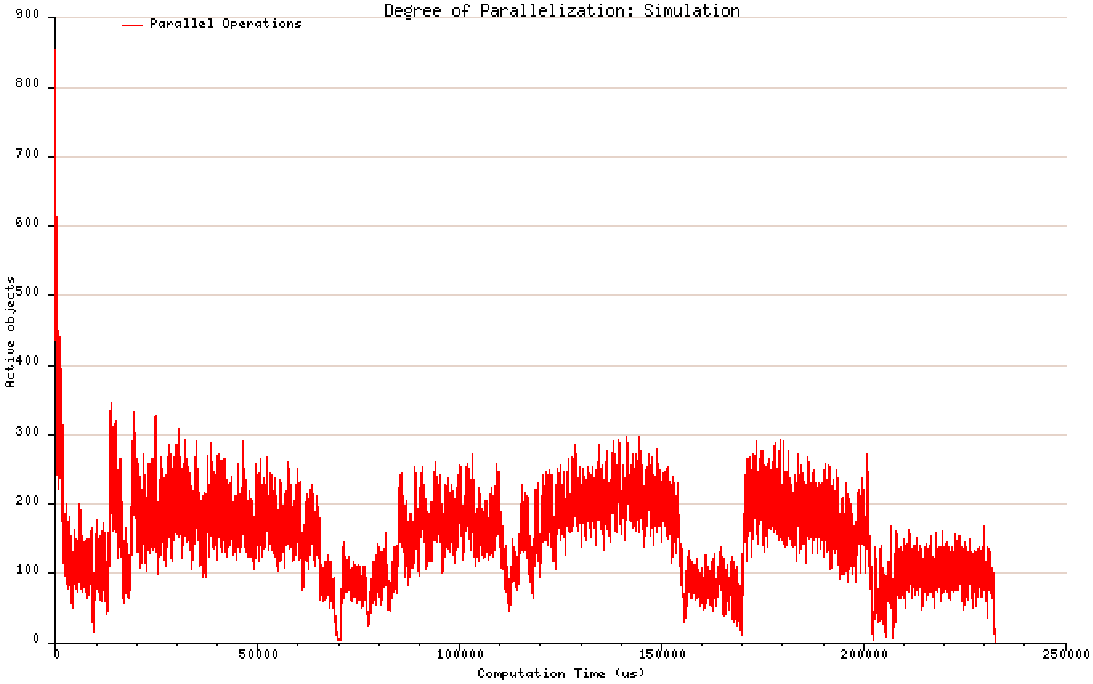


Figure 10.2: Degree of Parallelism of a 128-node interconnection network simulation implementation

This graph varies considerably from the first and is not what we would expect to see. We generate this graph in a similar manner to the previous graph, by logging the start and end times of events. However, in this case we log real time. All objects now have an *object real time* (ORT) associated with them which is stopped an restarted whenever an event is executed. Events are similarly timestamped with a *real-time stamp* which is the ORT of the object on which the event was generated at the point at which it was generated.

For example, suppose we have an object o. o starts with an ORT of 0.0. o receives an initial event at with real-time stamp 0.0. o starts to execute the event by calling a timer and recording its startTime t_s. o performs some work and is about to send an event, but first it calls the timer again to get the current time t_c. The event is timestamped ORT+$(t_c - t_s)$ and is sent. o continues to finish the event. At the end, it calls the timer again to get the end time t_e. It has just elapsed a total of $t_e - t_s$, so it advances its ORT by adding $t_e - t_s$ to it. The event is logged with a start and end time of the old and new values of the ORT.

When an object receives an event with a real-time stamp less than the current ORT, the real-time stamp is simply ignored. If, however, the real-time stamp is greater than the ORT, the ORT must be advanced to the event's real-time stamp. This advance is idle time on that object and thus is not logged.

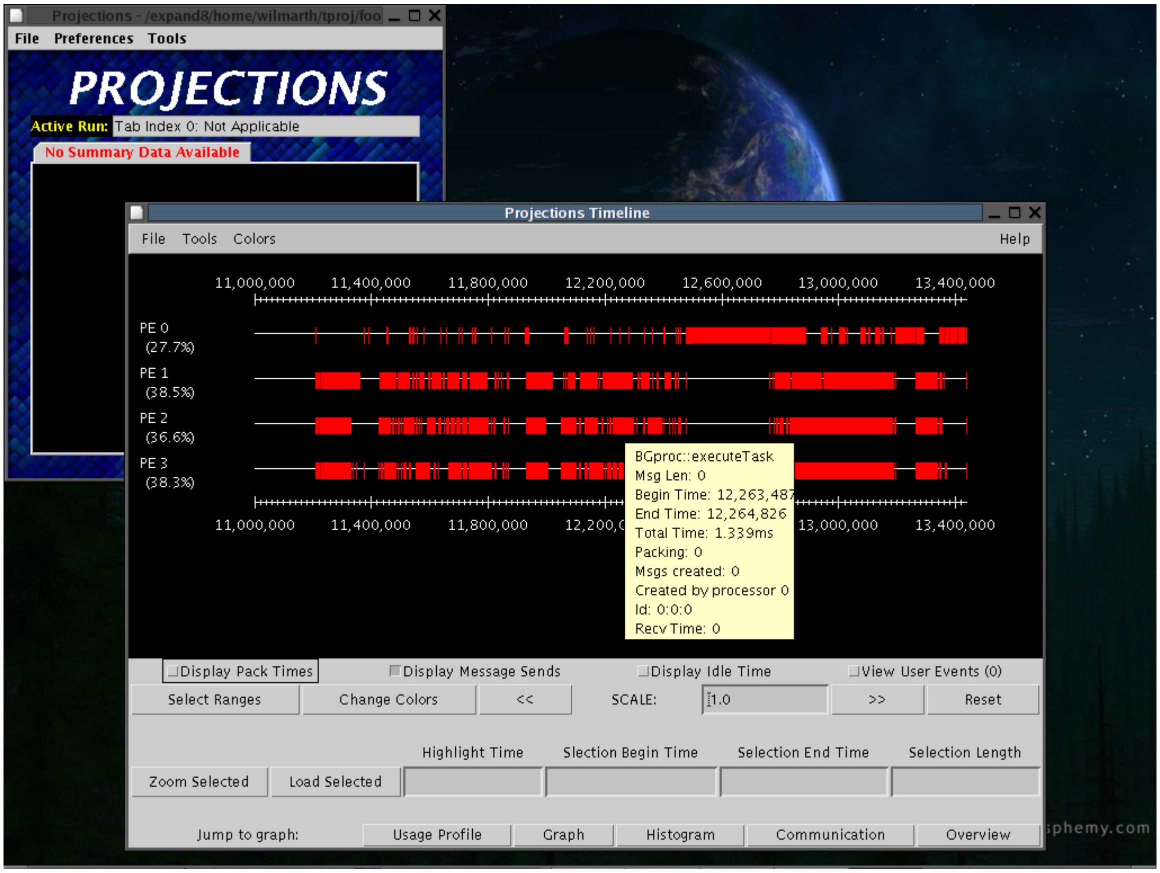
To generate the graph, event durations are again mapped onto bins according to real time units. Our current implementation uses microsecond bins.

Figure 10.2 shows us a considerably different view of the simulation. The first thing we notice is that the degree of parallelism is more even throughout the length of the program, averaging around 150 active objects, varying for the most part between 50 and 300. We also notice that there are phases where the number of active objects is much lower than average. Analyzing these variations could lead to insights into how to improve the performance of the simulation. We discuss network simulation in more detail in the next chapter.

10.2 Visualizing Model Performance

Projections[27, 25, 24] is a performance analysis tool for parallel programs. It presents the user with several possible views of program performance. Of particular interest to us is the timeline view. Timeline shows activity on each processor over time. Activity is in the form of events and each event is given a unique color and appears on the timeline as a bar, the length of which denotes the duration of the event. The view shows us how a parallel application behaves over time and can give insight into problems of load imbalance, idle time, dependencies, etc.

We leveraged the capabilities of Projections to visualize performance of the model being simulated by the Pose program. Projections reads its information about program behavior from log files that were created while the program was running. We have added a feature to Pose that logs information about PDES events on objects and their durations with respect to virtual time. When we read this data into Projections, instead of presenting one timeline per processor, we present one per object. The view in Figure 10.3 shows us the pattern of events on all the objects over virtual time (but "PE x" refers to object x, not a PE).



Active Run: Tab Index 0: Not Applicable

BGproc::executeTask
Msg Len: 0
Begin Time: 12,263,487
End Time: 12,264,826
Total Time: 1.339ms
Packing: 0
Msgs created: 0
Created by processor 0
Id: 0:0:0
Recv Time: 0

SCALE: 1.0

Figure 10.3: Projections display of model activity

This is particularly useful if we are simulating systems for which we are trying to derive an optimal configuration. Perhaps we are simulating industrial processes that require large pieces of expensive equipment in a factory, and we want to know the lowest cost configuration of equipment that will produce the maximum profit final result. We might find that a planned configuration is too large for the operation and that expensive components are idle more than 50% of the time. Or we might find that an additional resource improves throughput at a minimal cost. Whatever the case, the timeline gives us a detailed picture of the operations in a system model.

There are additional features of Projections that may also be of use here. We can view utilization of each entity over the course of the simulation in various ways. We can also view a total picture of the simulation in a single screen overview. This might show us patterns we never knew were there. For example, if we were modelling migratory patterns of birds over the years, we might notice that changes in habitat over time are changing the migration pattern.

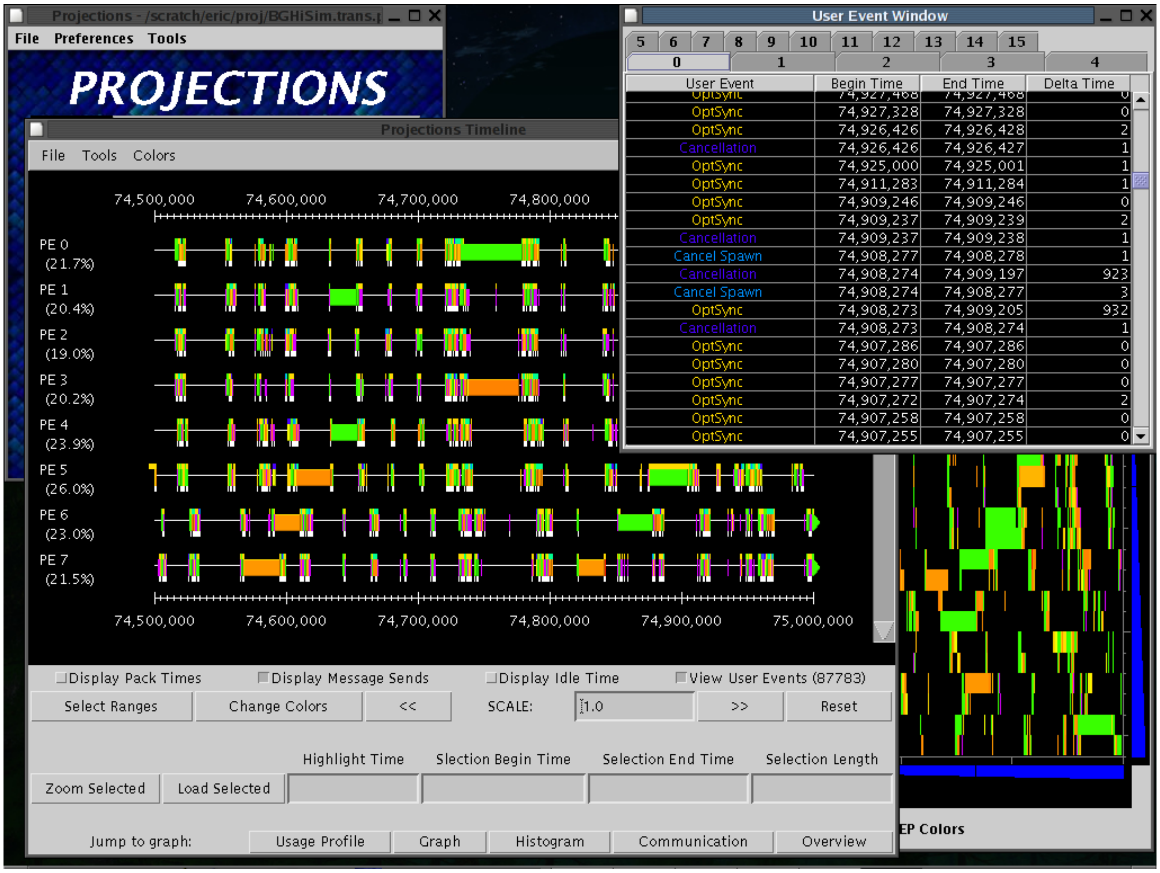
Projections was not designed for this sort of visualization, but PDES itself is highly amenable to Projections-style logging of information. We discuss future directions in PDES visualization in Chapter 12.

10.3 Visualizing Simulation Performance

In a similar manner, we can log events on objects according to their *real computation time* as opposed to their virtual time. With this information, we can visualize the simulation performance as a parallel application and see potential for improvement by viewing the timelines. In this case, a timeline represents how the simulation would look if we had infinite processors (or one per object) and no overhead. It shows the best possible parallel performance of the simulation as it was coded. It may give the programmer insight into why a particular simulation is not achieving good speedup or how a different initial placement of entities on processors might improve performance.

10.4 Visualizing Pose Performance: TRACE_DETAIL

A third form of simulation performance analysis can help the user decide how to configure Pose to work with specific applications. This approach breaks the activity of the simulation into its various components and lets the user visualize them with Projections views enhanced by the addition of *user events*. Projections allows application developers to add special events to regular Projections log files. These events show up in the Projections timeline as a thin bar above the regular timeline as shown in Figure 10.4. The figure shows one timeline per processor with real time along the x-axis.



User Event	Begin Time	End Time	Delta Time
OptSync	74,927,468	74,927,468	0
OptSync	74,927,328	74,927,328	0
OptSync	74,926,426	74,926,428	2
Cancellation	74,926,426	74,926,427	1
OptSync	74,925,000	74,925,001	1
OptSync	74,911,283	74,911,284	1
OptSync	74,909,246	74,909,246	0
OptSync	74,909,237	74,909,239	2
Cancellation	74,909,237	74,909,238	1
Cancel Spawn	74,908,277	74,908,278	1
Cancellation	74,908,274	74,909,197	923
Cancel Spawn	74,908,274	74,908,277	3
OptSync	74,908,273	74,909,205	932
Cancellation	74,908,273	74,908,274	1
OptSync	74,907,286	74,907,286	0
OptSync	74,907,280	74,907,280	0
OptSync	74,907,277	74,907,277	0
OptSync	74,907,272	74,907,274	2
OptSync	74,907,258	74,907,258	0
OptSync	74,907,255	74,907,255	0

Figure 10.4: Projections display of Pose activity

Projections is primarily for the display of Charm++ entry methods and much of the work in Pose is performed in forward executions that are not invoked directly by entry methods, but rather by components of the GVT algorithm. Thus Projections timelines for Pose simulations can appear to be comprised almost entirely of GVT algorithm code. Turning on the `TRACE_DETAIL` option creates logs which display this second timeline above the first. This timeline displays Forward Execution, Checkpointing, Rollback, Cancellation and Fossil Collection activities.

Chapter 11

Network Simulation for Large Parallel Architectures: A Low Degree of Parallelism Case Study

Many extremely large parallel machines have been built or are in progress, and many more are slated to be constructed in the near future. These machines will have incredible performance possibilities with very large numbers of processors capable of petaFLOPS-level peak performance. How will existing applications perform on such machines? Will they scale, or can they be made to scale by the time the machines become available?

The BigSim [55, 56, 53, 54] project, worked on by Gengbin Zheng and others at the Parallel Programming Laboratory, attempts to answer these questions. This project aims at developing techniques to facilitate the development of efficient peta-scale applications on very large parallel machines. BigSim allows for the simulation of the execution of key applications on large parallel machines, which may or may not have been built yet. The idea behind the project is to give application programmers the chance to test and predict performance of their applications on large parallel machines without actual access to the machines. This ability makes it possible to predict the performance of these machines before they are built for certain target applications. It also makes it possible to analyze and debug application performance glitches before using up limited time on existing machines which can be hard to come by and involve long waits. We have used Pose to implement a variety of simulations of networks for very large parallel machines to improve the accuracy of performance prediction for key applications. These network simulations comprise BigNetSim, a part of the BigSim project.

11.1 BigSim

BigSim performs the difficult task of performance prediction in a flexible manner, allowing for different levels of fidelity for components at the processor and network level. BigSim operates in two modes: an on-line latency-based simulation mode and an emulation mode which leaves the network modelling to post-mortem analysis code. Both modes provide for various approaches to modelling the execution of the application code: 1) User-specified execution time for each component; 2) Instruction-level simulation of the code on target architecture; 3) Performance counters; 4) Wall clock time multiplied by a CPU factor (to adjust simulation machine speed to target machine speed).

The on-line simulation executes the application in parallel with several simulated nodes modelled on each physical processor. Many parallel applications have a deterministic behavior that can be easily simulated without regard to ordering of computation components, so full simulation is not required. Instead, this mode uses a PDES-like approach to correcting the timestamps of computation events. On-line simulation models simple latency-based network models for BlueGene/C, BlueGene/L and QsNet (Lemieux).

The emulation executes the application code without correcting timestamps or modelling latency and logs information about each sequential computation block or *task* performed by the application to disk. These logs are then available for post-mortem analysis which is the role of the BigNetSim component of the project.

11.2 BigNetSim

BigNetSim encompasses a set of POSE simulations that read the logs from the BigSim emulation of a program. The logs consist of sets of tasks representing sequential computation blocks of the original application that was run on the emulator. Each task has a start time (which is based on the zero-latency emulation and will be corrected by BigNetSim), a duration, a set of dependencies, and a set of generated tasks and their offsets from the start time. This information is sufficient for performing detailed simulations of an application's behavior on a variety of network topologies and contention models.

This simulation illustrates the challenges to PDES well. The interactions between the entities in the network model primarily involve the transmission of packets through the model with very little computation on the entities themselves. Thus the grainsize of each event is extremely small at approximately $4\mu s$ on average. Further, there are vast numbers of events which introduce much overhead in synchronizing them, as well as a very large communication overhead. Another challenge is that applications seldom generate constant peak traffic on a network, so significant portions of the network are idle, leading to a very low degree of parallelism. The strong dependencies between events in a network with application-generated traffic also contribute to a low degree of parallelism.

We first implemented a simple, variable, latency-based simulator that we would later be able to plug contention-based network models into.

11.2.1 Simple Latency-based Network Simulation

For simple latency-based network simulation, we read the log files generated by the BigSim emulator. The size of the log is proportional to the number of messages exchanged. An application execution was emulated on some configuration, and all the sequential computation blocks (called *tasks* in postmortem simulation), messages generated by these tasks, and the dependencies between tasks are recorded in these logs. In our simulation, we recreate entities

```
executeTask(task)
  if (task.dependencies = 0) //dependencies met
    oldStartTime := task.startTime;
    task.startTime := ovt;  //correct start time
    for each task y in task.generatedTasks
      yStart := task.newStartTime +
        (y.generatedTime - oldStartTime);
      generate executeTask event on y at time
        yStart+latency; // or pass to detailed net simulator
    end
    elapse(task.duration); //advance virtual time
    task.done := TRUE;
    for each x in task.dependents //enable dependents
      decrement x.dependencies;
      if (x.dependencies = 0)
        generate executeTask event on y at time ovt ;
      end
    end
  end
end
```

Figure 11.1: Task “execution” in simple latency-based network model.

in Pose to model the processors and nodes of the emulation. We then read in the tasks and use the simulation to pretend to execute them. For each task, we know what it depends on, what depends on it, the duration of the task, and what other tasks were generated by it and when these other tasks were generated (as an offset from the current task’s start time). We also 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.

What we do not know is exactly when each task started (though we do have an uncorrected timestamp for each task), and without that information, we do not know how the emulated application performed. Given the information above, we start the first task off at virtual time zero, and let the tasks “execute” and record the virtual time at which each task starts. The algorithm for this process is shown in Figure 11.1.

When a task executes, it first checks to make sure that all its dependencies have been met, i.e. all tasks on which it depends have been executed. If they have, then it is time to execute this task. We make a backup copy of this task's incorrect timestamp (for calculating offsets of generated tasks later) and record the processor's current virtual time (OVT) as the task's correct start time. Then we invoke `executeTask` for all of this task's generated tasks, calculating the start time for each by offsetting the correct start time for this task by the same offset as before and adding in the latency to send the message to its destination.

Next, we elapse the local virtual time by the duration of the task and mark it done. Now it is safe to enable any tasks that were dependent on this one. The algorithm goes through all the dependents, and if a dependent is enabled (it is not dependent on any other unexecuted tasks), it can be executed immediately.

When all tasks have been executed, they should have correct timestamps and the final GVT should represent a correct runtime for the emulated application.

11.2.2 Detailed Contention-based Network Simulation

Detailed contention-based network modelling takes BigNetSim to the next level. Instead of using a preset latency value to determine message transit time, we actually model messages as they pass through a detailed contention-based network model. The power of this approach is that we can model any type of network we wish and plug it into the postmortem simulation to get new results. This enables us to run the application emulation once and reuse the logs generated by the emulation to repeatedly analyze the application in a variety of network configurations.

LoSim

LoSim (Low Fidelity Simulation) is our first, "fixed" approach to the problem of modelling a detailed contention-based network. It uses a virtual cut-through-like routing scheme on a direct torus network topology in which nodes talk to switches which are connected via command and data channels. Command channels are used to establish locked paths between source and destination nodes. Once established, a message is packetized and sent along the path to the destination. The path is freed up as the last packet of the message passes over each link. Contention is modelled as messages collect at switches and wait to be sent and packets intersect and are routed through the switches.

HiSim

HiSim (High Fidelity Simulation) [15], developed by Praveen Kumar Jagadishprasad and others at the Parallel Programming Laboratory, is a modular set of simulation components that can model a variety of architectures such as BlueGene, Hypercube and RedStorm. It supports pluggable modules for topology (3D mesh, torus, hypercube, fat tree) and routing strategies (adaptive, static) and will eventually support hybrids of direct and indirect networks of torus and fat tree configurations.

It contains Pose components for processors and nodes as in the latency-based simulation, but also models Net Interfaces, Switches and Channels. It includes a TrafficGen component for generating network traffic that can be used to test the network model without running actual applications through BigSim and relying on log files. Figure 11.2 illustrates the posers associated with a single BlueGene node in a HiSim simulation.

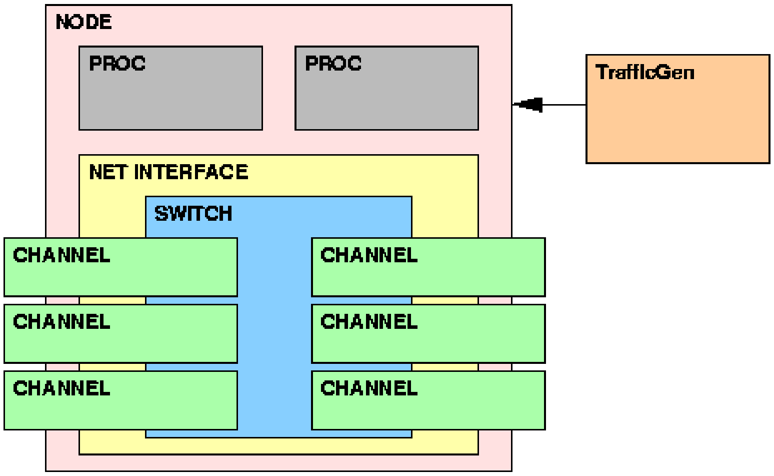


Figure 11.2: HiSim poser objects for a single BlueGene node

The HiSim network performance prediction environment facilitates informed decision making regarding the design of high speed interconnects. As such it provides a configurable runtime environment where network parameters, such as buffer size, number of virtual channels, packet size, adaptive vs fixed routing, etc. can be set at run time. Furthermore, the design is modular to support easy extensions by the addition of new topologies, routing algorithms and architectures.

When evaluating a new network, or proposed changes to an existing network, it is essential to be able to study network performance based on the expected traffic loads. The user may choose to load the network with application-generated tasks from log files or create network load patterns using the TrafficGen interface.

In studying the performance of a network via simulation, it is useful to consider the spatial distribution of messages in an interconnection network. These distributions can be represented by some standard traffic patterns which are used to evaluate interconnection networks. These patterns are based on communication patterns that arise in specific applications. HiSim includes the TrafficGen module to generate such patterns.

TrafficGen is a module that sends messages from a node to a destination node that is picked according to a predefined pattern. The traffic generator is implemented as a poser. Each node in the network being simulated has a TrafficGen poser associated with it. Six standard traffic patterns can be simulated with the traffic generator. They are:

1. k-shift: This is a cyclic shift by k wherein the node P_i sends a message to the node $P_{(i+k) \bmod (N)}$.

2. Ring: Each node sends a message to the next node, equivalent to a 1-shift.

3. Bit transpose: The address of the destination node is a transpose of that of the source node. i.e. $d_i = s_{(i+b/2) \bmod (b)}$ for the i^{th} bit in a b-bit address.

4. Bit reversal: Address of the destination node is a reversal of the bit address of the source node i.e. $d_i = s_{b-i-1}$.

5. Bit complement: Address of the destination node is a bitwise complement of the address of the source node.

6. Uniform distribution: This implies random traffic in which each source is equally likely to send to each destination.

The departure time between messages can be deterministic or follow a Poisson distribution.

11.3 BigNetSim Performance

We first take a look at how POSE handles network simulation with artificially generated traffic via the TrafficGen module. Then we examine both the simple latency-based network simulation and the detailed contention-based network simulation using traffic generated by a real application.

11.3.1 Performance using TrafficGen

Using our detailed network contention model configured to simulate the direct 3D torus network of the BlueGene/L machine. We experimented with a uniform distribution of random traffic generated by the TrafficGen module. These experiments were performed on Tungsten.

Figure 11.3 shows that POSE scales well relative to single processor parallel POSE in spite of the fine granularity of the events (average 0.000004s). We were unable to obtain a sequential POSE run for this problem, and so we have shown additional plots on this graph relative to T_e, estimated sequential time and T_i, ideal sequential time.

To get a better idea of how well POSE can adapt to the scale of the problem, we ran similar experiments with our network size ranging from 4x4x4 to 16x16x32. We ran each experiment on up to 64 physical processors and counted the number of hops (a hop is a single packet transmission from one node to another) and the number of POSE events. We

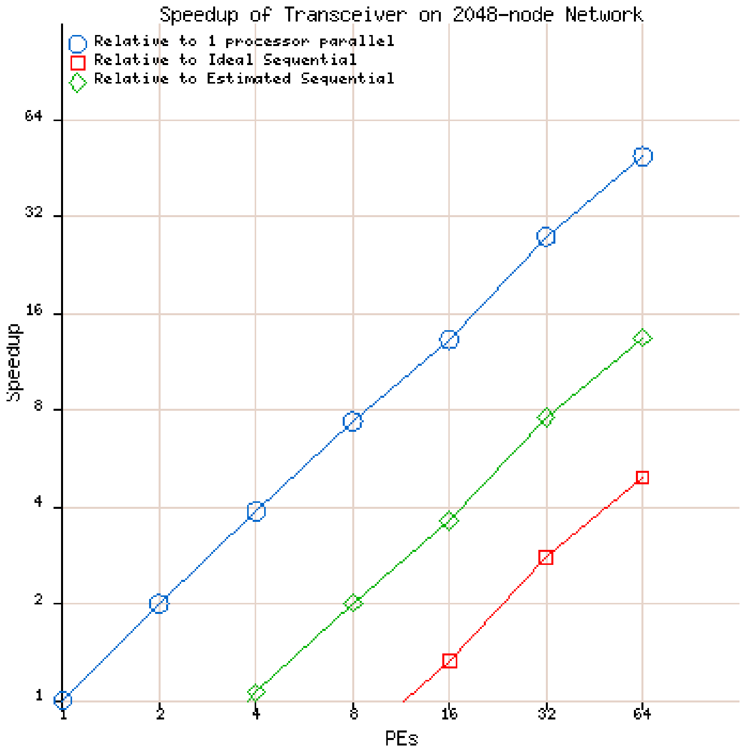
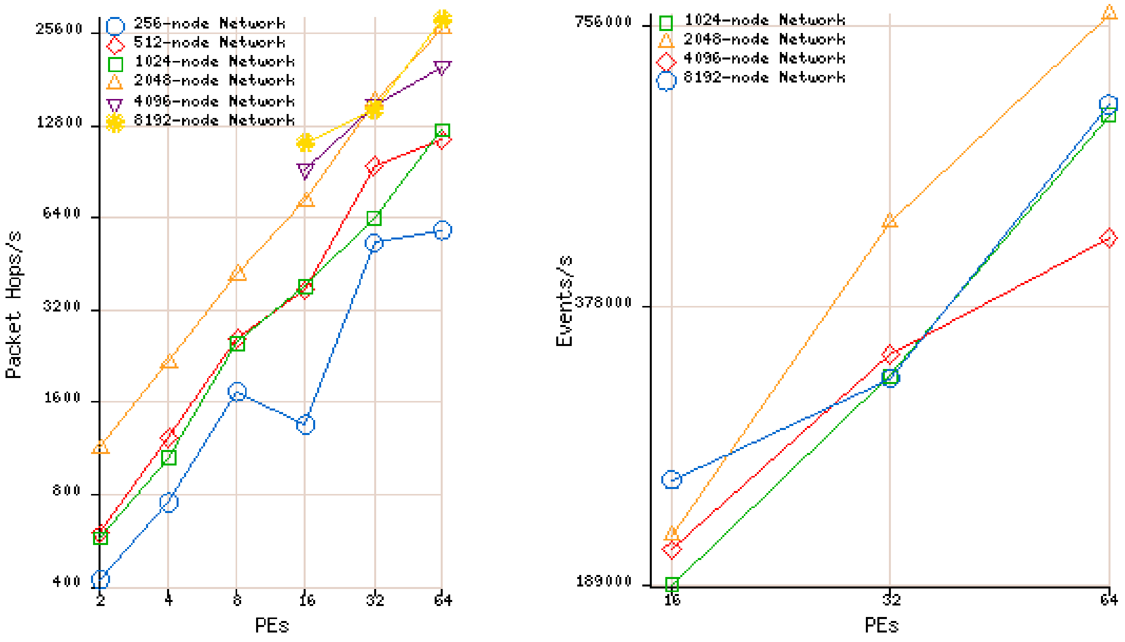


Figure 11.3: HiSim with TrafficGen Speedup

are concerned here with being able to handle twice as much work when we have twice the resources (processors) for simulation. Here we specifically examine the number of hops simulated per second of real time. Since there are many POSE events involved in setting up a single hop in our detailed network simulation, we also examine the scalability of events per second.

As shown in Figure 11.4(a), the simulations scale well with respect to how quickly they manage packet transmissions, and we roughly see a doubling of hops per second when we double the number of physical processors the simulation was run on. Our smallest network exhibited some erratic behavior, but larger networks scaled very well. For the largest networks we were not able to run as many tests by this writing, though they do show increases in hops per second of at least 50% in response to a doubling of physical processors.

Our detailed contention-based network simulator currently generates in excess of 50



(a) hops/second

(b) events/second

Figure 11.4: Problem scalability in HiSim with TrafficGen

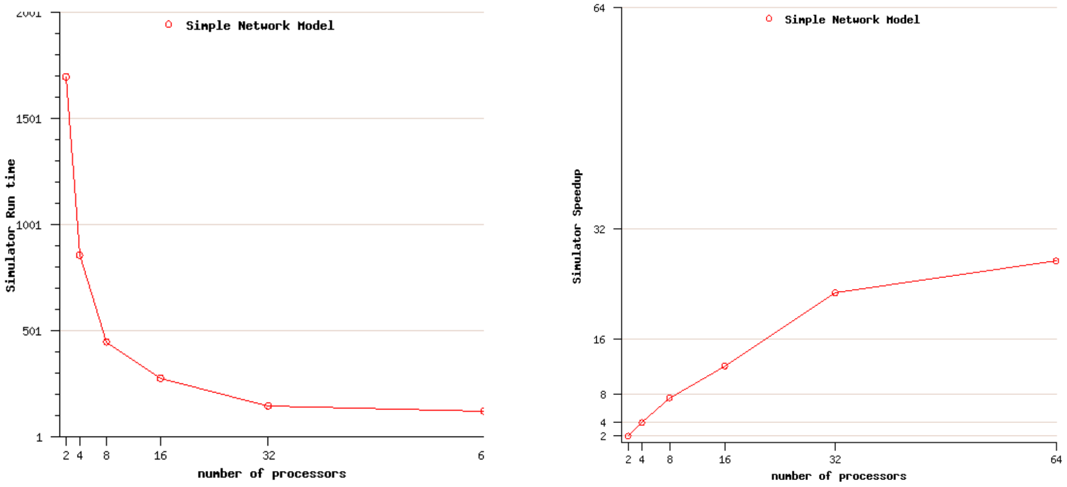
events per packet hop on average. Figure 11.4(b) shows a graph of the number of events committed by Pose per second on average for the same set of runs as in Figure 11.4(a), again doubling the physical processors to show that on average the events committed per second are also nearly doubled. This is significant considering that we are not including the events that were executed but subsequently rolled back and possibly never committed. At the peak, we achieve 780267 committed events per second on 64 processors for the 2048-node network.

11.3.2 Performance using Application-generated Traffic with Latency-based Network Simulation

We have evaluated our simple latency-based simulator for some real applications for large numbers of processors. One challenging application is Molecular Dynamics simulation of biomolecules. It is one of the important applications for BlueGene/L and other large parallel machines. The application we chose is NAMD [38], which is the current state-of-art high performance molecular dynamics code designed for large biomolecular systems.

As a NAMD benchmark system we used Apo-Lipoprotein A1 with 92K atoms. The simulation runs for a total of 15 timesteps. A multiple time-stepping scheme with PME (Particle Mesh Ewald) involving a 3D FFT every four steps is performed. We used the BigSim emulator on 64 real processors to run the NAMD program on 2048 simulated processors. The log files generated by the emulation were then read by our simple latency-based network model simulator using a varying number of processors. The NAMD 2048 processor run generates trace logs of approximately 700MB in total size which makes it challenging to simulate sequentially especially if the processor used has a small amount of memory.

We show a simulation execution time plot for the POSE simulation from 2 to 64 processors in Figure 11.5(a) and a corresponding speedup plot in Figure 11.5(b). The graph shows that the network simulator scales almost linearly to 32 processors and keeps scaling to 64 processors.



(a) Execution time

(b) Speedup

Figure 11.5: Performance of Latency-based Network Simulation with NAMD on 2048 Processors

This experiment shows that the Pose latency-based network simulator performs well even when applied to performance prediction of a real world application. This is a more challenging problem than network simulation with artificial traffic due to complex dependencies among events resulting in relatively limited parallelism in the simulation. However, simulating at the packet-level with network contention is even more challenging as we shall see in the next section.

11.3.3 Performance using Application-generated Traffic with Detailed Contention-based Network Simulation

We ran the same NAMD simulation of the previous section with our detailed contention model and the performance was dramatically different. Unfortunately, the performance degradation prevented most runs from completing. The degree of parallelism is very low making it difficult to select a number of processors to run the simulation on. Selecting a number of processors low enough to achieve a decent amount of parallelism often does not have enough collective memory to manage the entire simulation. Yet the artificial traffic generated by TrafficGen resulted in simulations with very good performance. This mystery is studied further in the next section.

LoSim, our first network simulator, uses a much less detailed contention model than HiSim. We were able to run simulations with NAMD-generated traffic and achieve some modest speedups relative to single processor parallel time as shown in Figure 11.6.

11.4 Analyzing the Degree of Parallelism of Detailed Network Simulation

In trying to determine why our detailed network simulator performs badly, we used some of the analysis tools described in the previous chapter. We returned to using TrafficGen so that

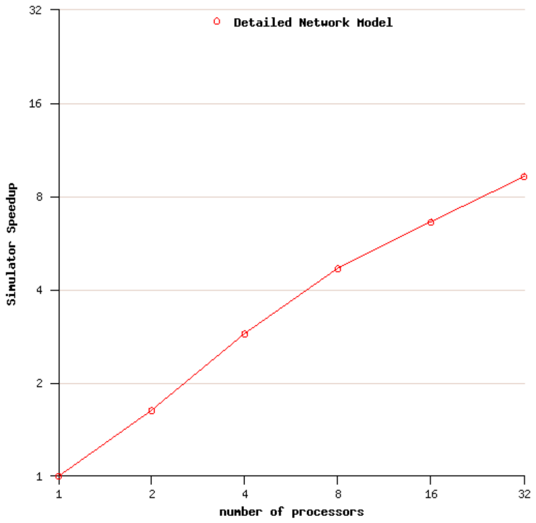


Figure 11.6: Speedup of LoSim Contention-based Network Model with NAMD-generated traffic

dependencies would be less of an issue, and so that we could flood the entire network with messages and see if any problems arose. We used a network size of 128 for our first experiment and graphed the degree of parallelization for both the model and the implementation.

What we discovered was that the model looked very much like we would expect it to, with large quantities of work generated at the beginning that die out through the course of the simulation. This is shown in Figure 11.7.

What we were not expecting was an unexplained sequential section from 7500 to 22500μs in the DOP graph for the *implementation* of that model in Figure 11.8. What this indicates is that there is some string of events being executed on one processor (discovered by examining the log files used to create the graph) on which the subsequent burst of activity depends, and that all work before this section either died out quickly or converged at this point.

This behavior was quite mysterious but certainly explained much of the poor performance results, particularly when we examined what happened on a larger network with 1024 nodes (Figures 11.9 and 11.10). In this case the convergence of events occurred at several points throughout the simulation, rendering a nearly 50% sequential program!

Next, we tried the 128 node simulation again but turned up the message density to 5

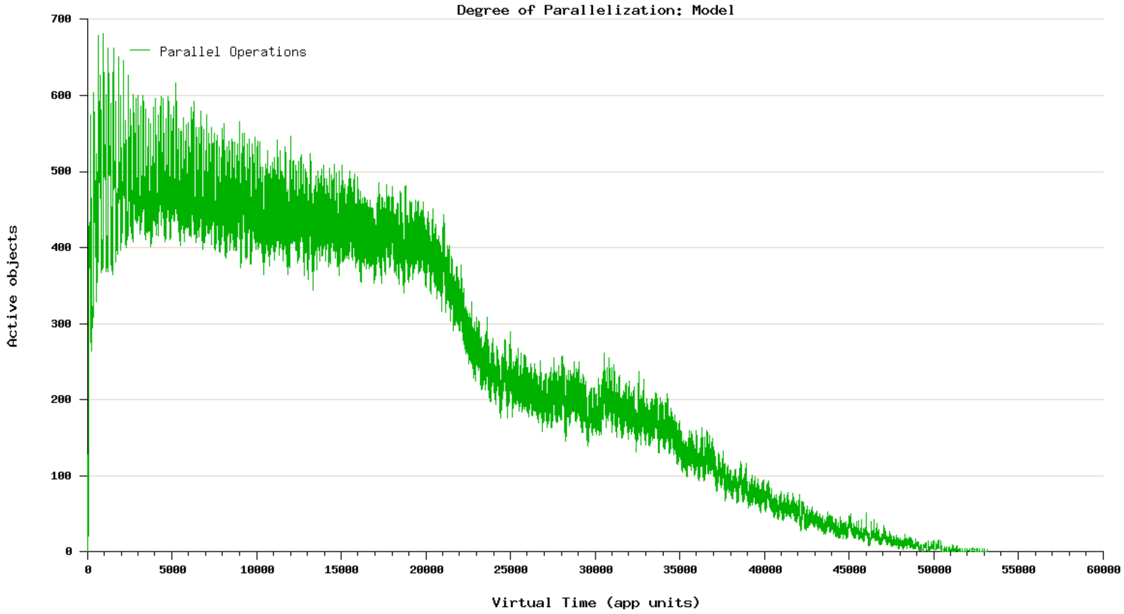


Figure 11.7: DOP for Detailed Contention-based Network Model with 128 nodes

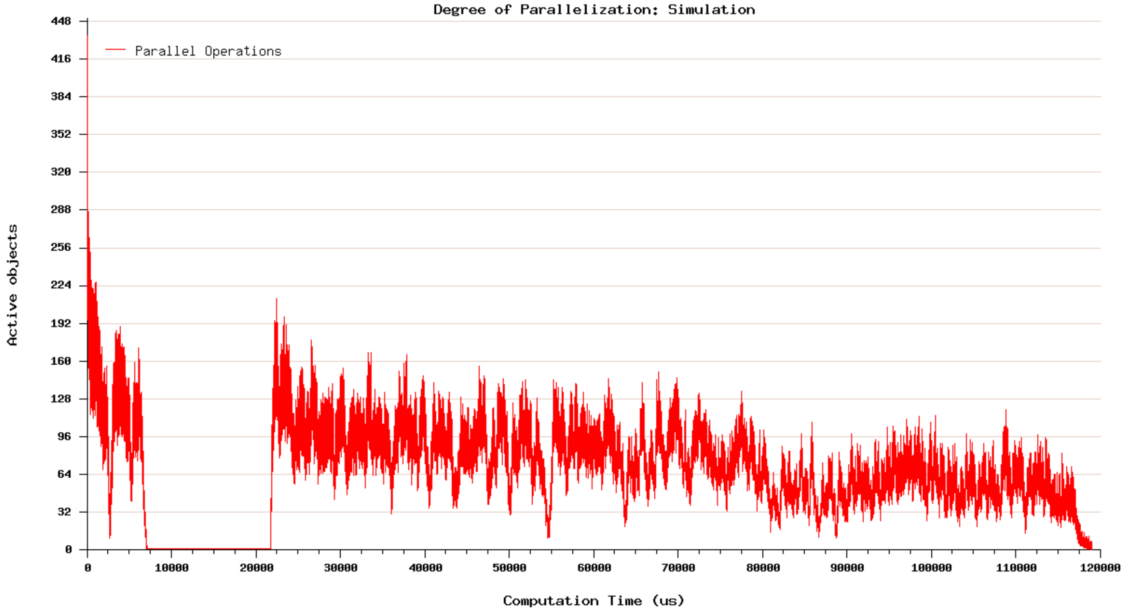


Figure 11.8: DOP for Implementation of Detailed Contention-based Network Model with 128 nodes

times what it was in the first case. Again, Figures 11.11 and 11.12 show mysterious sections of very low parallelism, but at least with a very high message density there was the possibility

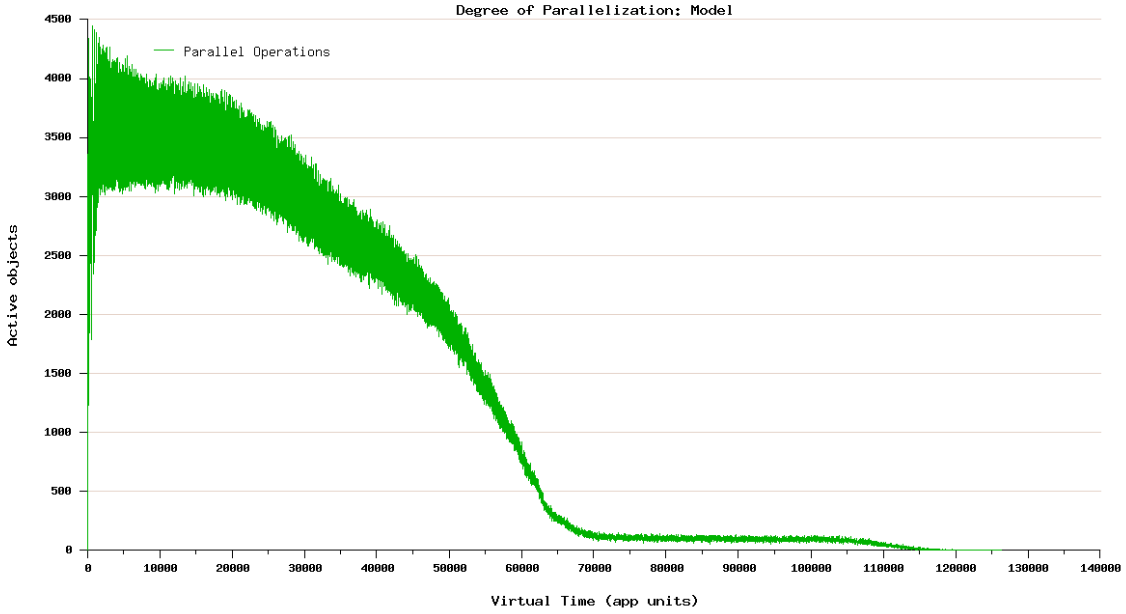


Figure 11.9: DOP for Model of Detailed Contention-based Network of 1024 nodes

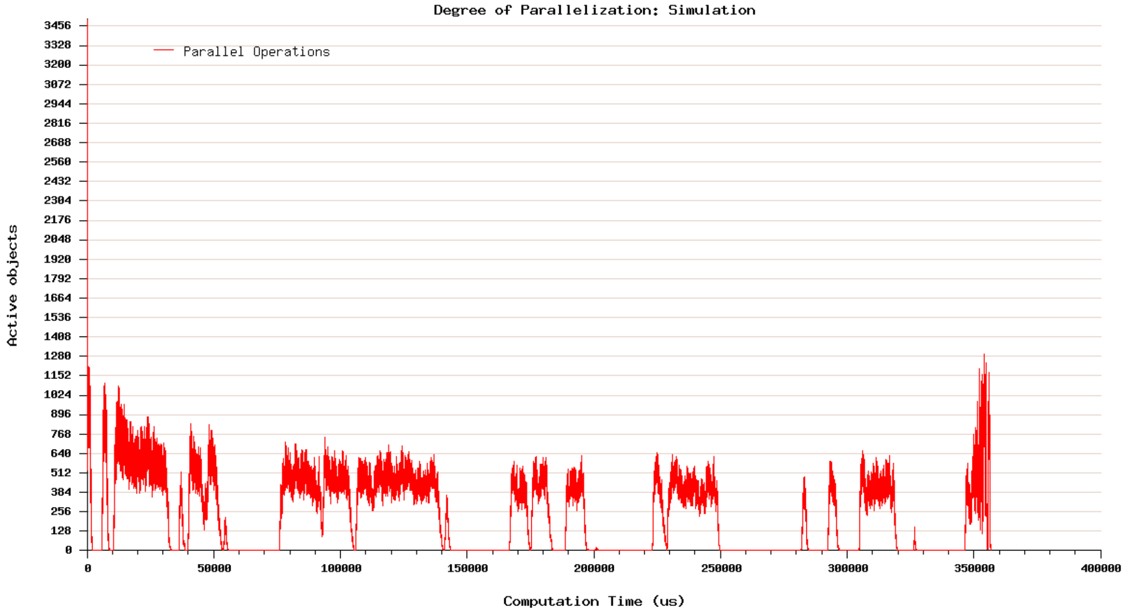


Figure 11.10: DOP for Implementation of Detailed Contention-based Network of 1024 nodes

for some overlap.

Application-generated traffic, with its dependency-ridden nature, cannot possibly perform well in such an environment. We have theorized that the sequential sections of code

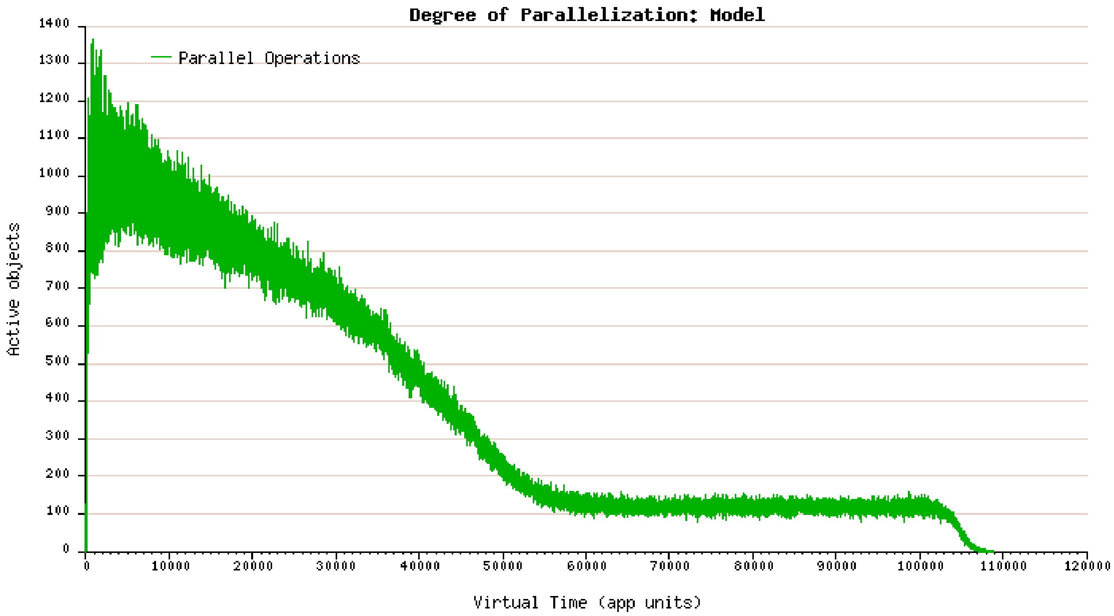


Figure 11.11: DOP for Model of Detailed Contention-based Network of 128 nodes with dense messaging

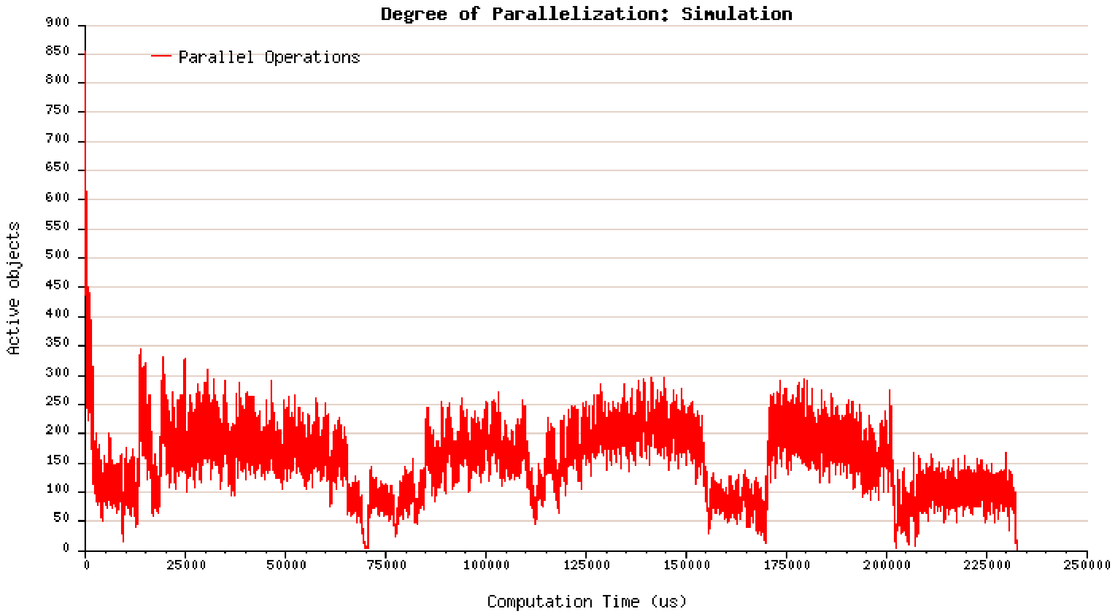


Figure 11.12: DOP for Implementation of Detailed Contention-based Network of 128 nodes with dense messaging

could be poorly optimized events that take considerable real time to compute but elapse very little virtual time and thus do not show up in the model DOP graphs. It is also possible that the routing algorithm is flawed or improperly implemented and eventually routes all packets through one particular node (or through some small set of nodes). Whatever the case may be, Pose has also proved to be a useful analysis tool for the developers of such simulations.

Chapter 12

Future Work

12.1 Usability

The current Pose implementation uses a source-to-source translator to convert Pose code into Charm++ (see Figure 12.1). This can make debugging awkward since line numbers refer to the Charm++ source instead of the original code. We plan to integrate Pose syntax into a new version of the Charm++ interface translator `charmc` called `posec`, which will remove this level of indirection.

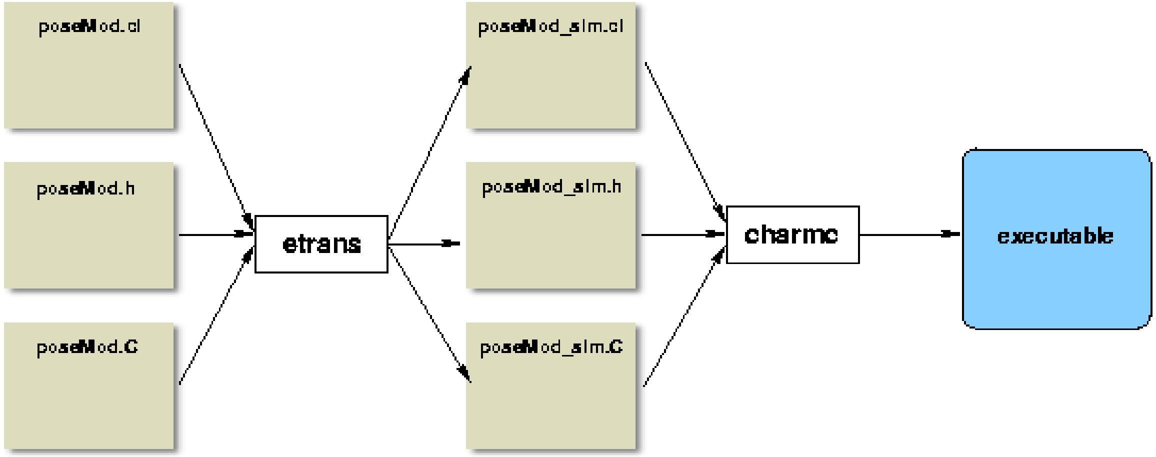


Figure 12.1: Source-to-source translation of Pose to Charm++

Along with this change, much about the syntax of Pose should improve as well, moving it closer to Charm++. For example, we should be able to provide the following features in a more straightforward manner:

- Parameter marshalling could be made to work for event methods. Thus events would look more like regular function calls. We would need to decide on a syntax for specifying an offset for the event.

- `void` events could be allowed. Currently, if there are no user arguments to a method, the user still needs to pass a base `eventMsg` type which can contain timestamp and other information.

- Strategy and representation specifications in the `.ci` file will no longer be necessary. The adaptive synchronization strategy is designed to replace all others.

- The Charm++ PUP Framework[37] will be used more extensively for checkpointing as well as migration. This framework implements sizing, packing and unpacking operations for data structures that serialize data for various purposes.

- Auto-generation of optional code such as pups, anti-methods and commit methods.

- Improved error checking should be possible.

- Inheritance.

Another future feature we hope to add to Pose is a Structured Dagger-like [18] syntax for specifying event dependencies on an object. This type of orchestration can be used to both improve the expressive power of Pose as well as improve performance. As an example, consider an entity that always receives two types (a and b) of events in pairs, and must process both parts of a pair before it handles any part of the next pair. We could specify this behavior as an event dependency, so that if and event of type a is received, it can be executed, and then if another a is received, the object does not execute it, but instead waits for an event of type b. This type of control allows the programmer to express assertions about the state of an object before the object handles particular events. With this addition Pose will be able to avoid the "risky" errors described in [36]. Currently, the burden of handling such situations is on the programmer, but orchestration will make it possible avoid these situations and thereby the overhead of handling them. For now, if the programmer detects a problem, an error state can be flagged, but the simulation does not terminate on an error condition until the flagged event is committed. Thus detection of error states and can be handled and if rollbacks undo the errors, the simulation can complete correctly.

Another usability enhancement involves moving most of the POSE configuration parameters out to the command line or configuration file level. Several of POSE's configuration specifications are currently in a header file in the POSE source, and thus changes to the configuration require a rebuild of POSE. We hope to be able to move these parameters out without adversely affecting POSE's performance.

12.2 POSE Production Version

The usability enhancements discussed in the previous section will greatly improve POSE's status as production-quality software. We have a wish list of several additional features we think would further enhance POSE.

12.2.1 Visualization and Analysis Capability

In Chapter 10, we discussed how we can analyze the performance of a model any type of system using Projections. We would like to expand on this capability in the future. In particular, when visualizing the timelines of objects, it would help if we could view the state of the object along the timeline as well. This might help explain the behaviors that we see along a timeline, and aid in determining how to improve the model to handle adverse situations.

Another enhancement to the visualization system we would like to pursue is the display of event dependencies on the timeline. Projections already has this ability for CHARM++ entry methods. To make use of it we need to add more detail to the information we are currently logging. The view will be optional, as it is now. The user will simply right-click on an event block, and a line will be drawn to the event that caused it.

We would also like to enhance the output capabilities of Pose by adding the ability to print information from committed events to a file. `CommitPrint` only prints to `stdout` and having an `FCommitPrint` would be very useful.

Another feature we wish to add is a `POSE_rand` function. This function will use the built-in rand() to generate a pseudo-random number during the first execution of an event. This number will be stored as a part of the state of the object at that point. If the event is rolled back and re-executed, we want the behavior to stay the same, so the when POSE_rand is called again, it simply gets the stored random number rather than generating a new one.

12.2.2 Functionality and Correctness

Pose originally had only integer timestamps, but now has a `POSE_TimeType` which can be switched from 32-bit integers to 64-bit integers. We plan to examine approaches to "infinite" timestamps, which will allow simulations to use very fine time units and run longer simulations without running out of time steps.

On a similar note, events have unique identifiers which wrap around and reuse earlier event IDs. This works for most applications since the earlier event IDs are no longer "alive". However, there may be very large simulations for which the set of "live" event IDs at any given time exceeds the possible unique event IDs available.

Another future goal for Pose is to implement existing algorithms for common problems that can arise in PDES. One example of this is the cancelback algorithm [17] for having an event consumer issue cancellations of events coming from an event producer. Without this approach the programmer must be careful that objects that do nothing but continuously produce events have some sort of control in place to prevent overflow on the receiving object.

We would also like to provide additional debugging utilities in Pose. Another problematic situation is when an object expects events to happen in a particular order and does not handle the error situations that can occur from handling an event that it is not ready for. Pose already provides a conditional error reporting mechanism, `CommitError` that only prints the error message if the event producing it gets committed. However, certain other erroneous situations can arise that are harder to handle. If an event executes on a state that it was not meant to execute on, an infinite loop may result. Some mechanism for infinite loop preemption is thus necessary. A more elegant way to handle these situations was already mentioned in the previous section however. Specifying event orderings in advance will eliminate the need for time-consuming error detection code.

12.2.3 Time and Space Efficiency

Pose and optimistic PDES applications in general are notorious for their large memory consumption. Checkpoints and other bookkeeping functions take up portions of memory until fossil collection can reclaim it. It takes considerable effort to keep the memory footprint of a simulation to a minimum. Pose has a mechanism in place for coarse management of checkpoints that enables a reduction of the memory footprint, but that mechanism could use further refinement. In particular it does not distinguish the variance in memory usage from one object to the next. A more flexible approach could restrict memory usage in a better way that does not hamper progress as much.

Another problem related to memory usage is that checkpoints are messages are continuously being allocated and deallocated. Because the memory deallocated was likely not recently used, it also takes longer to deallocate. With messages, we have implemented a memory recycling facility which places messages that are no longer needed into a pool that can be reused. This mechanism could use improvement and a similar mechanism should be employed for checkpoint recycling.

12.3 Application Libraries

There are many application areas which would benefit from special purpose libraries that can be used to design new simulations. We are in the process of making the entities in our network contention simulation more modular so that they can be reused to build future network simulations. We would also like to explore the field of circuit simulation by directly creating libraries of components to model and simulate any circuit.

Chapter 13

Conclusion

The question has been posed of parallel discrete event simulation: Will the field survive?[11] The answer is: Yes. Simulation is a valuable tool for analyzing the world around us. The more we learn, the more detailed and complex simulations become. They also become more demanding on resources, taking up more space in memory and running for longer periods of time. To meet that demand, we must continue to develop new techniques. Efficient PDES may be a challenging problem, but it is partly because of that challenge and the demands of simulation that people will continue trying to improve the technique.

This thesis has explored radical new directions in optimistic synchronization for PDES as well as more pragmatic and traditional approaches for improving PDES performance. We first developed Pose, an environment in which to experiment with our ideas, and to that we added the notion of virtualization: that every little entity in a simulation could be a logical process on its own. With this came a localization of all the baggage that accompanies optimistic synchronization: checkpointing, straggler detection, rollbacks, cancellations, fossil collection. This meant that when an error occurred on one of these small entities, only the entity itself need be affected. Small entities have less state to checkpoint, and they do so at a rate corresponding to the rate at which only that entity receives events to process. When they rollback, they only rollback the events they received, and they only cancel events they generated. The Charm++ run-time system supports such virtualization by allowing the execution of thousands of LPs per processor without the overhead associated with conventional implementations, such as processes or threads.

With the benefits of the reduced overhead made possible by this model, each individual object is now free to *take more risk*, executing more events speculatively. Entities now execute events out of timestamp order *on the same processor*. Having this ability further makes it possibly to perform several events in a row as a *multi-event* thereby reducing scheduling overhead and improving cache performance. Furthermore, each entity has its own instance of the scheduling strategy and can *adapt* how it speculates to its particular behavior.

On the more practical side, we have also explored communication optimization and load balancing for PDES. We have demonstrated how streaming communication optimizations can greatly improve the performance of PDES. We have also shown that the nature of PDES is such that load balancers must take multiple different factors into consideration compared to traditional load balancing practices. We have built a load balancing framework into Pose and demonstrated with a simple strategy that object progress — not idle time — is more important to balance.

Throughout this thesis, we illustrate our approaches with a fine-grained synthetic benchmark that allows for a wide variety of parameters to control its behavior. Using this benchmark, we have shown that our strategies perform very well, provided the degree of parallelism is good. Pose has also been used extensively to implement a variety of interconnection network simulators for large parallel machines and shows much promise for handling these challenging low-parallelism situations. It has also proved itself useful as an analysis tool.

We believe that we have contributed to the viability of the field of PDES in this thesis by developing novel approaches to the optimistic synchronization mechanism. We believe optimistic synchronization holds the most potential for scaling PDES to larger numbers of processors and that Pose has laid the groundwork for future research into scalable optimistic PDES.

References

[1] R. Bagrodia, R. Meyer, Takai 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 and W. T. Liao. Maisie: A language and optimizing environment for distributed simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 205–210, 1990.

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

[4] Rajive L. Bagrodia and Wen-Toh Liao. Maisie: A language for the design of efficient discrete-event simulations. *IEEE Transactions on Software Engineering*, 20(4):225–237, April 1994.

[5] Robert K. Brunner. Versatile automatic load balancing with migratable objects. TR 00-01, January 2000.

[6] Myongsu Choe and Carl Tropper. On learning algorithms and balancing loads in time warp. In *Workshop on Parallel and Distributed Simulation*, pages 101–108, 1999.

[7] Samir R. Das. Adaptive protocols for parallel discrete event simulation. In *Winter Simulation Conference*, pages 186–193, 1996.

[8] Samir Ranjan Das, Richard Fujimoto, Kiran S. Panesar, Don Allison, and Maria Hybinette. GTW: a time warp system for shared memory multiprocessors. In *Winter Simulation Conference*, pages 1332–1339, 1994.

[9] Ewa Deelman and Boleslaw K. Szymanski. Dynamic load balancing in parallel discrete event simulation for spatially explicit problems. In *Workshop on Parallel and Distributed Simulation*, pages 46–53, 1998.

[10] Alois Ferscha and Satish K. Tripathi. Parallel and distributed simulation of discrete event systems. In A. Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*, pages 1003 – 1041. McGraw-Hill, 1996.

[11] Richad M. Fujimoto. Parallel discrete event simulation: Will the field survive? *ORSA Journal on Computing*, 5(3):213–230, Summer 1993.

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

[13] Richard M. Fujimoto. *Parallel and Distributed Simulation Systems.* John Wiley & Sons, Inc., 2000.

[14] Albert G. Greenberg, Boris D. Lubachevsky, and Isi Mitrani. Algorithms for unboundedly parallel simulations. *ACM Transactions on Computer Systems (TOCS)*, 9(3):201–221, 1991.

[15] Praveen Kumar Jagadishprasad. Parallel simulation of large scale interconnection networks used in high performance computing. Master's thesis, University of Illinois at Urbana-Champaign, 2004.

[16] 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.

[17] David Jefferson. Virtual time ii: Storage management in conservative and optimistic systems. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 75–89. ACM Press, 1990.

[18] L. V. Kale and Milind Bhandarkar. Structured Dagger: A Coordination Language for Message-Driven Programming. In *Proceedings of Second International Euro-Par Conference*, volume 1123-1124 of *Lecture Notes in Computer Science*, pages 646–653, September 1996.

[19] L. V. Kale, Milind Bhandarkar, and Robert Brunner. Run-time Support for Adaptive Load Balancing. In J. Rolim, editor, *Lecture Notes in Computer Science, Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun - Mexico*, volume 1800, pages 1152–1159, March 2000.

[20] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

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

[22] Laxmikant V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.

[23] Laxmikant V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[24] Laxmikant V. Kalé, Sameer Kumar, Gengbin Zheng, and Chee Wai Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In

Terascale Performance Analysis Workshop, International Conference on Computational Science(ICCS), Melbourne, Australia, June 2003.

[25] Laxmikant V. Kale, Gengbin Zheng, Chee Wai Lee, and Sameer Kumar. Scaling applications to massively parallel machines using projections performance analysis tool. In *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, number to appear, 2005.

[26] L.V. Kale, B. Ramkumar, V. Saletore, and A. B. Sinha. Prioritization in parallel symbolic computing. In T. Ito and R. Halstead, editors, *Lecture Notes in Computer Science*, volume 748, pages 12–41. Springer-Verlag, 1993.

[27] L.V. Kalé and Amitabh Sinha. Projections: A preliminary performance tool for charm. In *Parallel Systems Fair, International Parallel Processing Symposium*, pages 108–114, Newport Beach, CA, April 1993.

[28] V. Kumar, A. Y. Grama, and Nageshwara Rao Vempaty. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22(1):60–79, 1994.

[29] T. Lai and T. Yang. On distributed snapshots. *Information Processing Letters*, 25:153–158, 1987.

[30] O. Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. In *Proceedings of ACM 2001 Java Grande/ISCOPE Conference*, pages 21–29, Stanford, CA, Jun 2001.

[31] Orion Sky Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. *Concurrency and Computation: Practice and Experience*, 15:371–393, 2003.

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

[33] Yi-Bing Lin. Determining the global progress of parallel simulation with fifo communication property. *Information Processing Letters*, 50(1):13–17, 1994.

[34] Yi-Bing Lin and Edward D. Lazowska. A study of time warp rollback mechanisms. *ACM Trans. Model. Comput. Simul.*, 1(1):51–72, 1991.

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

[36] David Nicol and Jason Liu. The dark side of risk (what your mother never told you about time warp). In *Workshop on Parallel and Distributed Simulation*, pages 188–195, May 1997.

[37] Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL. *The Charm++ programming language manual, Version 5.8 (Release 1)*, 2004.

[38] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.

[39] Bruno R. Preiss. The Yaddes Distributed Discrete Event Simulation Specification Language and Execution Environment. In *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, pages 139–144, March 1989.

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

[41] P. Reiher and D. Jefferson. Virtual time based dynamic load management in the Time

Warp operating system. In *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, pages 103 – 111, 1990.

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

[43] Jeff S. Steinman. Interactive speedes. In *Proceedings of the 24th annual symposium on Simulation*, pages 149–158. IEEE Computer Society Press, 1991.

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

[45] Jeffrey S. Steinman. Scalable parallel and distributed military simulations using the speedes framework. In *ELECSIM*, 1995.

[46] Alexander I. Tomlinson and Vijay 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.

[47] Krishnan Varadarajan. Communication library for parallel architectures. Master's thesis, Dept. of Computer Science, University of Illinois, 1999. http://charm.cs.uiuc.edu/papers/KrishnanThesis99.html.

[48] Jerry Waldorf and Rajive Bagrodia. Moose: A concurrent object oriented language for simulation. 0(0):0–0.

[49] Terry Wilmarth and L. V. Kalé. Pose: Getting over grainsize in parallel discrete event simulation. In *2004 International Conference on Parallel Processing*, page (to appear), August 2004.

[50] Linda F. Wilson and David M. Nicol. Automated load balancing in speedes. In *Proceedings of the 27th conference on Winter simulation*, pages 590–596. ACM Press, 1995.

[51] Linda F. Wilson and David M. Nicol. Experiments in automated load balancing. In *Proceedings of the tenth workshop on Parallel and distributed simulation*, pages 4–11. IEEE Computer Society Press, 1996.

[52] Linda F. Wilson and Wei Shen. Experiments in load migration and dynamic load balancing in speedes. In *Proceedings of the 30th conference on Winter simulation*, pages 483–490. IEEE Computer Society Press, 1998.

[53] Gengbin Zheng, Gunavardhan Kakulapati, and Laxmikant V. Kalé. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, New Mexico, April 2004.

[54] Gengbin Zheng, Arun Kumar Singla, Joshua Mostkoff Unger, and Laxmikant V. Kalé. A parallel-object programming model for petaflops machines and blue gene/cyclops. In *NSF Next Generation Systems Program Workshop, 16th International Parallel and Distributed Processing Symposium(IPDPS)*, Fort Lauderdale, FL, April 2002.

[55] Gengbin Zheng, Terry Wilmarth, Praveen Jagadishprasad, and Laxmikant V. Kalé. Simulation-based performance prediction for large parallel machines. In *International Journal of Parallel Programming*, number to appear, 2005.

[56] Gengbin Zheng, Terry Wilmarth, Orion Sky Lawlor, Laxmikant V. Kalé, Sarita Adve, David Padua, and Philippe Guebelle. 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)*, Santa Fe, New Mexico, April 2004. IEEE Press.

Vita

Terry Wilmarth was born in Dunkirk, New York and grew up in the small town of Fredonia. She graduated as valedictorian of her class of 1987 from Brocton Central School. With the aid of several scholarships, she attended the State University of New York at Buffalo and completed her B.S. in Computer Science in 1991 with honors. Continuing at SUNY Buffalo, she earned a M.S. in Computer Science in 1993.

Terry continued her studies at SUNY Buffalo for another two years, completing all Ph.D. qualifying hurdles, but decided to transfer to the University of Illinois at Urbana-Champaign in 1995 for a change of pace. She joined the Parallel Programming Laboratory in 1996 and began her studies with Prof. L.V. Kalé.

At PPL, Terry began her research on languages, developing a Parallel Object-based Language and an Actor Language in CONVERSE. She assisted Milind Bhandarkar with a Parallel version of Java. Subsequently, she worked on seed-based load balancers for Converse, developing many strategies for the initial placement of parallel objects. She developed an interest in parallel discrete event simulation based on previous work at PPL and decided to pursue that as her dissertation topic. She has simultaneously been employed as a research assistant for the Center for Simulation of Advanced Rockets where she has been working on parallel mesh adaptivity frameworks for 2D and 3D unstructured meshes.

After completion of her PhD, Terry will be seeking employment in academia.