Proceedings of the 2006 Winter Simulation Conference
L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds.

ELIMINATING REMOTE MESSAGE PASSING IN OPTIMISTIC SIMULATION

David W. Bauer Jr.

The MITRE Corporation
7525 Colshire Drive
McLean, VA 22102, U.S.A.

Christopher D. Carothers

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180, U.S.A.

ABSTRACT

This paper introduces an algorithm for parallel simulation capable of executing the critical path without *a priori* knowledge of the model being executed. This algorithm is founded on the observation that each initial event in a model causes a *stream* of events to be generated for execution. By focusing on the parallelization of event streams, rather than logical processes, we have created a new simulation engine optimized for large scale models (i.e., models with 1 million LPs or more).

1 INTRODUCTION

Since Chandy and Misra (1979), Misra (1986), and Jefferson and Sowizral (1982), parallel discrete event simulation has primarily been built around the concept of event list management. A logical process (LP) creates and sends events to other LPs in the system. LPs are then mapped to processing elements (PEs) for parallelization. Creating a parallel simulation executive capable of generating good speedup in the execution of generic models has been somewhat more elusive.

As the number of processors used in the computation is increased, the relative performance of simulation engines decreases due to the overhead of parallelizing the model. Several high performance simulation engines have reduced this overhead significantly (Carothers, Bauer and Pearce 2002; Das et al. 1994; Nicol 1988, 2002; Preiss 1989; Riley 2003; Syzmanski et al. 2003). However, when compared to linear, the performance is generally lacking for difficult benchmarks such as PHOLD (Fujimoto 1990).

The critical path of events for a model identifies the opportunities that exist for parallelizing the model. Simulators that attempt to generically parallelize models have always been confounded when it comes to parallelizing the critical path, because it varies from model to model. Several techniques have been proposed by (Berry and Jefferson 1985, Lin 1992, Zhou et al. 2002) for critical path analysis.

Amdahl's law states that for the fraction f of computation that *cannot benefit from parallelization*, the maximum speedup obtainable converges to $1/f$. Events sent remotely between processors characterize areas of a model that must be serialized. We propose a parallel discrete event algorithm that generically and automatically parallelizes the critical path as defined by the model mapping of LPs to PEs. Our main observation is that in large-scale models, as additional processing power is applied to a model, speedup in typical event list managed simulators *must decrease* as the remote event rate increases. By eliminating contention due to remote events we expect to achieve a more efficient parallelization for models.

Computer architecture is quickly expanding making the idea of *personal supercomputers* a reality. Advanced Micro Devices (AMD) and Intel have released their first versions of dual-core processor technology in the Opteron and Xeon families, respectively. Soon quad-core processors will become available, and motherboards to support multiple quad-core processors. In addition, HyperTransport technology (Wikipedia 2006) is enabling manufacturers to connect multiple motherboards within a single computing platform via a high bandwidth, low latency bus. The supercomputing culture was changed with the advent of (primarily) Linux-based cluster computing platforms. These technologies will transform the culture again by overcoming the high latencies that continue to exist in high performance networks.

Our results are gathered on a quad-processor dual-core Opteron system, and show that for large-scale models (i.e., models with greater than 1 million LPs) an improvement can be gained using our approach. We compare performance against ROSS (Carothers, Bauer, and Pearce 2002) and show an improvement of ∼29% in a 10 million LP PHOLD benchmark.

In this paper we introduce a new algorithm for the parallelization of the critical path in its general form by eliminating the passing of events between remote processors. In the following sections we will define the problem (Section 2), outline the algorithm applied to the problem

(Section 3), and provide a detailed performance analysis of the Adirondack stream simulator versus ROSS (Section 4). We follow-up with comments for related work, future work and a conclusion (Sections 5, 6 and 7). We have affectionately named our implementation *Adirondack* for the numerous streams contained in the Adirondack State Park in New York.

2 PROBLEM DEFINITION

As additional processors are utilized in a parallel system, overhead due to parallelization increases. In ROSS, the only locks that appear in the scheduler are located around event (including cancel event) sends, and in the synchronization algorithm. For synchronization, ROSS, GTW and Adirondack all employ Fujimoto's algorithm for shared memory multiprocessors (Fujimoto 1990) and are implementations of the optimistic synchronization algorithm. This algorithm executes asynchronous with event execution and the global virtual time (GVT) algorithm does not become a performance bottleneck. The traditional approach towards discrete event simulation, namely event list management, does however. Overhead waiting for contention to be resolved on the inbound event queue for a PE increases with the number of processors. Cancelation events typically comprise a small portion of the total events executed and are less important to optimize.

As an example, the benchmark model PHOLD selects the destination for an event from a uniform distribution around the number of total LPs in the model. Figure 1 shows that the probability of the destination entity being mapped to a remote processor increases logarithmically as the system is parallelized, with a limit at 100%. With the exception of embarrassingly parallel models, this behavior is exhibited in varying degrees by many models, and so the mapping of LPs to processors becomes a performance bottleneck.

Resolving this bottleneck typically involves model specific information to properly map the LPs to the PEs for the purposes of reducing the rate at which remote events are generated. This problem is generally left for the modeler to resolve. Contention between execution threads occurs during model execution most frequently when events are sent between processors. To address this inefficiency for all models requires *we take a different approach towards event list management, specifically, where contention is placed within the system.*

3 OVERVIEW OF STREAM SIMULATION

Stream simulation is an algorithm for parallelizing a discrete event simulator based on event streams, rather than passing events between remote processors. In a stream simulator, there are *zero remote events.*

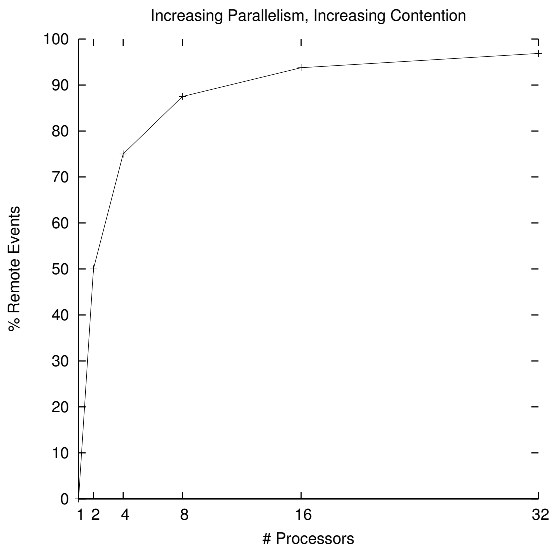

Figure 1: PHOLD Remote Event Rate Increases Logarithmically as Processors Are Added.

The problem of contention arises when a processor must stall, waiting for access to another processor's inbound event queue. While this is a relatively inexpensive operation, it falls into the common case for the execution of any model. One approach towards resolving this contention might be to aggregate events on the sending processor's side. When the queue becomes free, the local list of pending events could be sent. The problem with this and other similar approaches in general is that the frequency of rolling back would be increased due to events which have arrived "late" in the optimistic execution of an LP. Our solution is based on the observation that traditional Time Warp systems parallelize models based on the LP-PE mapping, rather than on the causal ordering of events that naturally occurs within any model.

An **event stream** is defined as an initial event created by a model, and all of the events that are *caused* caused as a consequence of that event. We emphasize "caused" to denote no logical correlation between the perceived meaning of an event in a model and the definition of an event stream. When a simulator executes an event, that result of that computation is zero or more new events being created in the model. If an initial event is ε, then the set of events that are created as a consequence form ε'. The set of events created as a consequence of executing all ε' events forms the set ε'', and so on until model termination.

Using the definition of an event stream, a stream simulator parallelizes a model by mapping event streams to processors for execution. Because an event stream is processed entirely within a single processor, *zero events are remotely passed between processors.*

3.1 Parallel Algorithm

In a typical event list managed simulator, events sent between LPs are determined by the LP to PE mapping, and this mapping is defined by the model. When two LPs communicate and are mapped to different processors, the corresponding event is considered to be a remote event. Events are passed between PEs, and the local causality constraint (Lamport 1978, 1979) is enforced within the domain of the PE, namely through event lists. In a typical shared memory Time Warp implementation, such as ROSS (Carothers et al. 2002) or GTW (Das et al. 1994), PEs must obtain a lock on an inbound event queue on the remote PE before adding that event to the list. Each PE is then responsible for adding those events to a local priority queue that causally orders the events for processing by that PE. All other simulator facilities are handled in a similar manner (i.e., fossil collection and rollback).

In a stream simulator, parallelization is correlated to event streams, rather than PEs. Events never migrate between remote processors in a stream simulator. Rather, for each event initialized on a PE, that event and all successive events in that stream will remain local to the PE to which it was mapped by the model. We must continue to ensure that the causality constraint is preserved by each PE as each event is processed. Generally, this means that each event's destination LP must be locked prior to execution, and the LP rolled back if necessary to ensure the event is executed in the correct time-stamp order.

Usually, this approach would lead to inefficient execution, as contention around LP states could be long-lived. In the case of large-scale models however, this is not the case. In fact, models where the number is CPUs is relatively low (less than 100) and the number of model LPs is high (greater than 1 million) yields a very low probability of two or more processors attempting to execute events for the same LP at any time in the system. **In a stream simulator, as the model size grows, the probability that contention will occur decreases**.

To illustrate these probabilities, we must consider the event populations being simulated. For these purposes, PHOLD is a significantly difficult model and easily illustrated, because event destinations are uniformly distributed over the LP population.

In ROSS an event population has the possibility of generating contention in the system based upon the event destination. When an event's destination LP is mapped to a remote processor, as shown in Figure 2, contention may occur.

$$\mathcal{P}(\text{Contention}) = \alpha\left[1 - \frac{1}{\rho}\right] \qquad (1)$$

where ρ = total CPUS, and α = remote event rate.

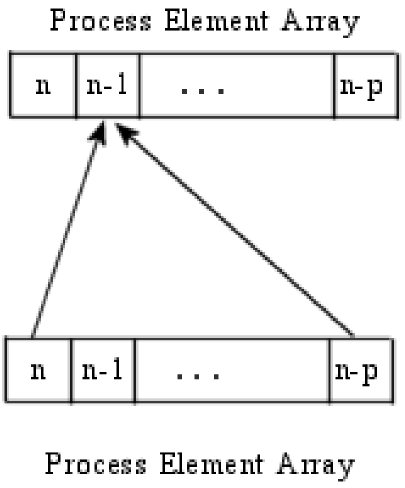

Figure 2: Contention in ROSS (η is the Number of LPs, and ρ is the Number of Processors).

Equation (1) states that as the number of CPUs utilized increases, the amount of contention for a fixed model increases. The α coefficient represents the amount of remote message passing for a given model. When $\alpha = 1$ (no remote events are sent by the model), then the probability of contention is zero. Conversely, when $\alpha = 0$, (uniform distribution of event destination LP over set of LPs), then nearly every event will be sent remotely and the probability of contention approaches 100%. Our algorithm applies in those cases where α is approaching one; we are concerned with models that exhibit high remote event rates. The PHOLD model exhibits the worst-case behavior and for the remainder of this discussion, α is implied.

In Adirondack, the event destination PE is independent of the executing PE. The probability of contention occurs around the LP state, and not the LP to PE mapping. Contention is determined by the probability that two or more processors are executing an event for the same destination LP only. The events dequeued for execution in the PE array define the size of the problem. Figure 3 illustrates the combinatorial nature of the problem. If the size of the LP array is η, and the size of the PE array is ρ, then the complete space is defined by η^{ρ}. Equation (3) illustrates that no contention occurs in those cases where each LP destination value is unique in the PE array. This is equivalent to the more general problem of determining the number of ways of obtaining an ordered subset of elements from a set of elements (Uspensky 1937). For example, there are $4!/2! - 12$ two-subsets of {1, 2, 3, 4}, namely {1, 2}, {1, 3}, {1, 4}, {2, 1}, {2, 3}, {2, 4}, {3, 1}, {3, 2}, {3, 4}, {4, 1}, {4, 2}, and {4, 3}.

Again assuming the worst case where all processors are perfectly synchronized, contention occurs when 2 or more processors are attempting to commit an event to the same

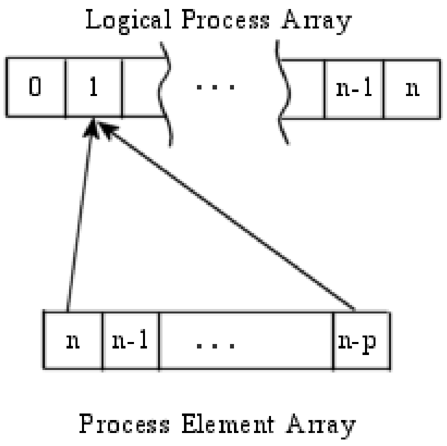

Figure 3: Combinatorial Nature of Contention in Adirondack.

LP. From this, the probability of contention be defined by the equations:

$$\mathcal{P}(\text{Contention}) \;=\; 1 \;-\; \frac{\mathcal{P}(\text{No Contention})}{\mathcal{P}(\text{Total})}, \tag{2}$$

$$\mathcal{P}(\text{No Contention}) = \;_{\eta}\mathcal{C}_{\rho} = \frac{\eta!}{(\eta-\rho)!} \tag{3}$$

where η = total LPs, and ρ = total CPUs,

$$\mathcal{P}(\text{Total}) \;=\; \eta^{\rho}, \tag{4}$$

$$\mathcal{P}(\text{Contention}) = \;1 \;-\; \frac{\frac{\eta!}{(\eta-\rho)!}}{\eta^{\rho}}. \tag{5}$$

Our conclusion is that as the model size grows, contention decreases in a stream simulator for large scale models where $\eta \gg \rho$. The complete relationship is given by Equation (6).

$$\mathcal{P}(\text{Contention}) = \begin{cases} \text{approaching } 0 & \text{if } \eta \gg \rho, \\ 1 & \text{if } \eta <= \rho, \\ 0 < \mathcal{P} < 1, & \text{if } \eta > \rho \end{cases} \tag{6}$$

Figure 4 is plotted for $\rho = 8$ to illustrate that this function is decreasing. The probability of contention is undefined for the sequential case. Equation (6) converges slowly to zero, and for $\eta = 10,000$, the probability of contention is less than 0.003%, and this formulation is valid for large-scale models.

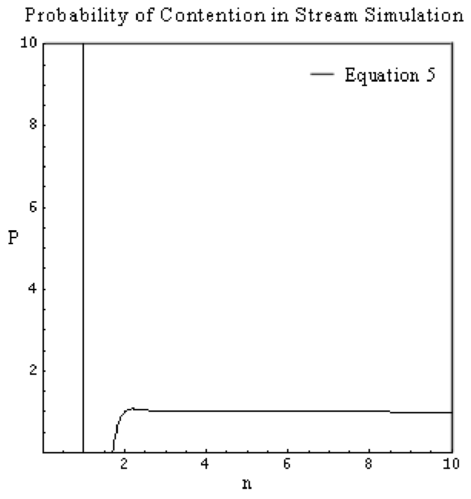

Figure 4: Equation (5), Plotted for $\rho = 8$ (Probability of Contention is 1 for $\eta = 1$, and Decreasing for $\eta > 1$).

3.2 Kernel Processes

Adirondack was developed from the ROSS source code. Consequently, both simulators contain an additional data structure for aggregating LP fossil collection, the *kernel process* (KP). Typically, LPs are mapped by the model to PEs. In these simulators, LPs are mapped to KPs, and KPs to PEs. For reverse computation, event states, rather than LP states are maintained for each GVT epoch. The events are stored in a processed event queue within the KP. Having a lower number of KPs relative to the number of LPs leads to efficient fossil collection by reducing the number of processed event queues that must be searched. Kernel processes were introduced in (Carothers, Bauer, and Pearce 2002).

There is a trade-off in Adirondack between a lower number of KPs for efficient fossil collection, and a higher number of KPs to reduce contention. We have previously shown that the overhead of fossil collection is a limiting factor in the scalability of ROSS. KPs were introduced as a scalable mechanism to reduce that cost by aggregating the LP's processed event queues. In the performance section we analyze the impact of varying the number of KPs in Adirondack to illustrate this trade-off.

3.2.1 Rolling Back

Rollbacks support optimistic execution within the Time Warp executive when the causality constraint is violated (Jefferson 1982). Optimistic execution allows each processor to commit events as quickly as possible, and then rollback those events when an out of order event (i.e., straggler event delayed in the system) is detected. Simulators like ROSS perform these tasks internal to a PE, and so only one CPU is ever operating on the LPs mapped to it.

Because Adirondack allows any processor to commit any event for any LP, straggler events may still occur. Straggler events happen when parallel streams cross at an LP and the streams are out of phase. Unlike ROSS, contention is introduced because LPs must be locked not only during the forward execution of events, but in the reverse computation of events as well. To avoid event streams from migrating across PEs, events canceled as part of a rollback still must be placed in the cancel queue of the PE that originally committed the event.

Because we can no longer rely on the PE priority queue to enforce the causality constraint, we must check for rollback just prior to event execution at an LP. This does not introduce additional rollbacks, rather, it changes the location where the causality constraint is examined.

3.2.2 Fossil Collection

Fossil collection also occurs at the LP level. Like ROSS, Adirondack utilizes reverse computation (Carothers, Perumalla, and Fujimoto 1999) rather than state-saving techniques to support rollback. Therefore, Adirondack saves events rather than LP states. These events must be routinely fossil collected in order for the model to continue advancing. After an event is committed by an LP, that event is then added to the processed event queue for that LP.

The asynchronous nature of the GVT algorithm allows for one processor to perform fossil collection while another continues to process events (or some other engine facility). Fossil collection then becomes a point of contention in a stream simulator. Fortunately, fossil collection of an LP's processed event queue can be done by the PE that LP is mapped to, and that PE only. While processor execution is asynchronous, fossil collection is a relatively synchronized event, tied to the GVT computation. These intervals should be sparse in relation to the number of events processed per GVT interval, and the cost amortized over the runtime.

4 PERFORMANCE STUDY

4.1 Computing Testbed and Experiment Setup

All experiments were conducted on a quad-processor, dual-core AMD Opteron server configured with 32GBs of RAM. The AMD Opteron 800-series chip enables 64-bit computing, and provides up to 24GB/s peak bandwidth per processor using HyperTransport technology. The DDR DRAM memory controller is 128-bits wide and provides up to 6.4GB/s of bandwidth per processor. Our RAM configuration consisted of 4GB sticks of 400MHz DDR ECC RAM in 8 banks.

The simulation executive had the following configuration parameters. Optimistic event memory was computed in each case by Equation (7).

$$\lceil \eta / \rho \rceil * \gamma * c. \tag{7}$$

Here, η is the total number of LPs, and ρ is the number of processors used for parallelization. Also, γ was the number of initial events per LP, was fixed at 1 for all experiments. We also have a constant factor c, that was fixed at 2.

GVT batch and interval parameters were set at 1024 and 8 respectively. Thus, up to 8,192 events will be processed between GVT epochs for both systems. These settings where determined to yield the highest level of performance for both systems on this particular computing testbed.

Because Adirondack was implemented from the ROSS source code, major facilities such as GVT computation, fossil collection, rollback, priority queues (Brown 1998) and random number generators (L'Ecuyer and Andres 1997) were identical. Additions to the scheduler including place mutexes around the rollback, fossil collection and event execution calls. Events were not sent remotely in Adirondack, so all event sends were simply enqueued directly into the calling PEs priority queue.

4.2 RC-PHOLD Model

The model used is a synthetic workload model called PHOLD. This commonly used benchmark has been modified to support reverse-computation and is configured to have minimal Logical Process (LP) state, message sizes and event processing. The forward computation of events involves computing two random numbers: one used to compute the time-stamp offset and one used to compute the destination LP for the next event in a stream.

Reverse computation involves "un-doing" an LPs random number generator (RNG) in order to restore its state. Because the RNG is perfectly reversible (LEcuyer and Andres 1997), the reverse computation restores seed state by computing the perfect inverse function as described in Carothers, Bauer, and Pearce (2002). The destination LP is determined by calling a uniformly distributed random number generator in the range of 0 to number of LPs - 1. The other call determines the offset timestamp for events and is exponentially distributed with a mean of 1.0; the model terminates at timestamp 100. For all experiment runs, we mapped LPs to Processing Elements (PE) in a round robin fashion. Each simulation run contained either 10 or 20 million LPs, and the number of Kernel Processes (KP) was varied from 1,000 to 1,000,000 by factors of 10. The message population per LP is 1.

This model is a pathological benchmark which has minimal event granularity while producing a configurable number of remote events which can result in thrashing rollbacks. In Carothers, Bauer, and Pearce (2002), KPs where introduced as an aggregation structure for reducing LP fossil collection and rollback.

4.3 RC-PHOLD Performance Data

The data for our initial performance comparison between Adirondack and ROSS using the quad processor dual-core Opteron server is presented in Figure 5. The event rate is shown as a function of the number of CPUs utilized for parallelization for both Adirondack and ROSS. The model is configured for 20 million LPs, and 100 thousand KPs for each experiment.

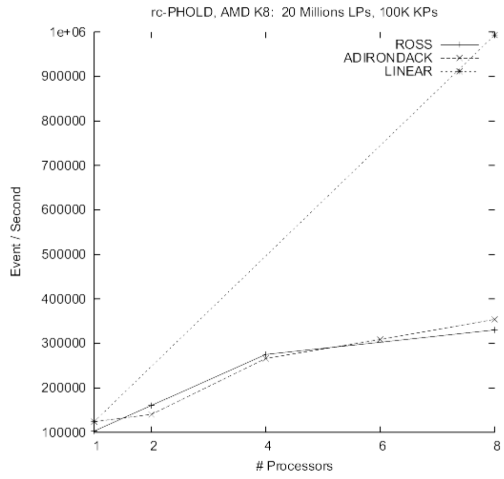

Figure 5: RC-PHOLD: 20 million LPs, 100,000 KPs.

It is important to note that on an eight processor system, we would likely achieve higher performance results in either simulator using fewer KPs. Because we are limited in the hardware available to us, we are attempting to model the system as though we had a much larger number of CPUs. A system with more CPUs would benefit from a larger set of KPs. As a consequence, fossil collection is not optimally efficient for the models presented here. Note that it is equally inefficient in both ROSS and Adirondack, so the comparisons are fair.

The performance for Adirondack begins to exceed that of ROSS as more processors added to the system, as expected. We did not observe a large difference in the performance for this model. Also, we were limited in the hardware available to us and and for our purposes a machine with 16 or 32 CPUs would have been more suitable. Finally, this server appeared to go into swap after allocating only 12GB of its available 32GB of RAM, so we could not go beyond 20 million LPs and 100 thousand KPs.

In Figure 6 we reduce the model size to 10 million LPs and 10 thousand KPs. ROSS performed about the same as in the previous model size. However, Adirondack showed a marked improvement, increasing performance by ∼29%.

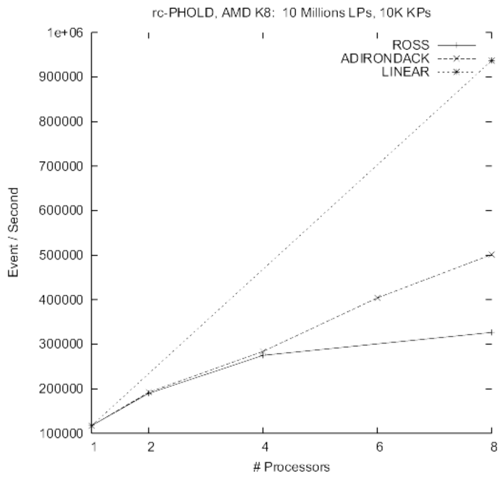

Figure 6: RC-PHOLD: 10 Million LPs, 10,000 KPs.

We believe that with a smaller number of KPs, Adirondack and ROSS are both able to perform fossil collection much more efficiently than before. However, ROSS still suffers from a high probability of contention, and so performance does not improve. Adirondack has a infinitesimally small probability of contention ($< 0.003\%$), and so performance increases in correspondence to the improvement in fossil collection. From Equation (1), the probability of contention in ROSS was 87.5%.

4.4 KP Performance

We now investigate the trade-off that occurs between fossil collection and contention. Kernel processes were developed as a method for aggregating fossil collection, but Adirondack defines contention at the KP level. A lower number of KPs is desirable for efficient fossil collection; a higher number of KPs is desirable for reducing contention.

Figures 7 and 8 illustrate the performance of ROSS versus Adirondack as the number of KPs are varied in the system. In the 4-CPU case, overhead due to contention is small in comparison to the amount overhead due to fossil collection, and so ROSS and Adirondack perform about the same. In the 1,000 KP experiment ROSS outperforms Adirondack significantly, because there are not enough KPs in the system for the stream parallel algorithm to operate efficiently.

In the 8-CPU case, contention begins to become more of an issue for ROSS in general, and again for Adirondack in the 1,000 KP experiment. As the number of KPs grows, and the number of processors grow, the stream parallel algorithm becomes more efficient and Adirondack outperforms ROSS.

Using either model, as the number of KPs approaches the same order of magnitude as the number of LPs, we begin to see the effects of inefficient fossil collection, as

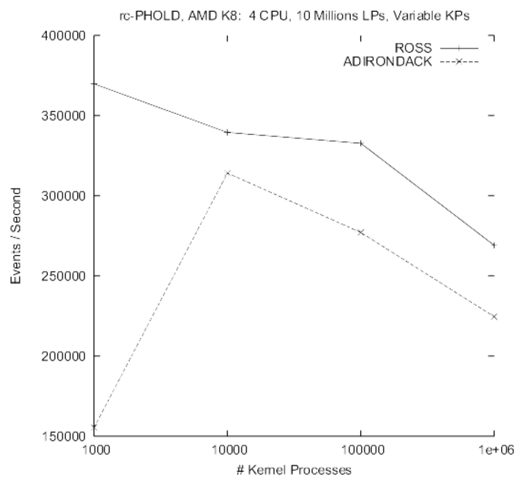

Figure 7: 4 CPU, Variable KPs.

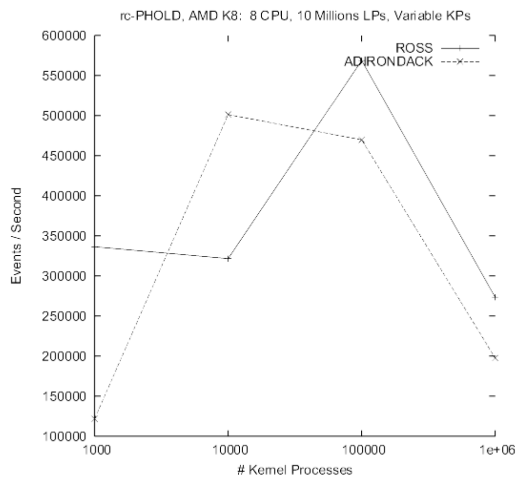

Figure 8: 8 CPU, Variable KPs.

observed in Carothers, Bauer, and Pearce (2002). As fossil collection dominates the computation, Adirondack drops below ROSS because of contention added around KP for the fossil collection routine.

There is a clear trade-off between minimizing contention and minimizing fossil collection overheads. We have shown that for significantly large scale models additional performance can be gained by achieving the proper balance of these two effects. In the 8-CPU, 10 million LP case, we had a performance gain of $\sim 29\%$.

We believe in a system with a higher number of CPUs (i.e., 16 or 32) the model partition per CPU decreases further, reducing the overhead of fossil collection per CPU. In addition, it is clear that a system with more CPUs would yield more contention, and benefit from this approach.

5 RELATED WORK

In Nicol and Liu (2002), a technique for composite synchronization of global and local protocols was introduced that relied on the classification of communication channels within the model. This technique characterized the model as a collection of sub-models, and clustered LPs into time lines for synchronization. Channels were classified as either synchronous or asynchronous, and time lines were allowed to “flow” across processors.

A global event queue was used in Prasad and Naquib (1995) to reduce rollback frequency and improve simulator efficiency for medium and coarse grained models. Similar to stream simulation, any processor was able to commit events for any destination LP, as long as that would not lead to contention. Contention was resolved using one of a parallel heap implementation (Prasad et al. 1994) or performing batch dequeue operations for all processors.

Srinivasan and Reynolds (1998) showed that *near perfect state information* protocols could outperform traditional Time Warps systems. They introduced the Elastic Time Algorithm that dynamically adapted to fit the model being simulated. This approach extends earlier ideas put forth by Reynolds (1988) who proposed that there are in fact multiple approaches to parallel and distributed simulation beyond conservative or optimistic methods.

In Zimmerman and Chandy (2005) an parallel algorithm for correlating streams of data is presented. Here, computational graphs are pipelined to achieve a highly efficient system that integrates multiple data streams. These systems are also referred to as *data fusion* systems and are applied to the problem of threat detection and opportunities.

6 FUTURE WORK

Future work would involve extending this parallel algorithm to a distributed cluster computing environment. Adirondack defines a stream as an initial LP event and the set of events caused by that event over the runtime of the model. We believe this definition could be extended logically to a distributed system where event streams could be initialized throughout the runtime. Then, events sent between remote processors on unique systems would create new event streams on the remote machine.

The trade-off of large numbers of KPs to support stream simulation hampers the benefits of KP aggregation in the fossil collection routines. We would like to investigate decoupling these facilities in some way within the system to achieve an efficient parallel algorithm that does not impact fossil collection. Doing so could yield a system with a broader range of performance.

7 CONCLUSIONS

We have identified a new parallel algorithm for discrete event simulation that executes the critical path of a model without *a priori* knowledge. We have implemented this algorithm in a simulator named Adirondack and shown how this new algorithm eliminates the passing of events between remote processors. In addition we have shown that the performance of this simulator is relatively equivalent to an existing state of the art optimistic simulation system, ROSS.

REFERENCES

Berry, O., and D. Jefferson. 1985. Critical path analysis of distributed simulation. In *Proceedings of the 1985 SCS Multiconference on Distributed Simulation*, 57-60.

Brown, R. 1988. Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM* 31(10):1220-1227.

Cai, W. T., and S. J. Turner. 1990. An algorithm for distributed discrete event simulation - the carrier null message approach. In *Proceedings of the SCS Multiconference on Distributed Simulation*, 3-8.

Carothers, C. D., D. Bauer, and S. Pearce. 2002. Ross: a high-performance, low memory, modular time warp system. *Journal of Parallel and Distributed Computing*.

Carothers, C. D., K. S. Perumalla, and R. M. Fujimoto. 1999. The effect of state-saving in optimistic simulation on a cache-coherent non-uniform memory access architecture. In *Proceedings of the 1999 Winter Simulation Conference*.

Chandy, K. M., and J. Misra. 1979. Distributed simulation: a case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440-452. 1979.

Das, S., R. M. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. 1994. GTW: a time warp simulator for shared memory multiprocessors. In *Proceedings of the 1994 Winter Simulation Conference*, 1332-1339.

Fujimoto, R. M. 1990. Parallel discrete event simulation. *Communications of the ACM* 33:31-52.

Fujimoto, R. M. 1990. Performance of time warp under synthetic workload. In *Proceedings of the SCS Multiconference on Distributed Simulation*, 22:1.

Fujimoto, R. M. 2000. Parallel and distributed simulation systems. In *Wiley Series on Parallel and Distributed Computing*, 51-95. Wiley-Interscience.

Jefferson, D. R., and H. Sowizral. 1982. Fast concurrent simulation using the time warp mechanism. Technical Report N-1906-AF, RAND Corporation.

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21:558-565.

Lamport, L. 1979. How to make a multiprocessor compute that correctly executes multiprocess programs. *IEEE Transactions on Computers* 28(9):690–691.

L'Ecuyer, P., and T. H. Andres. 1997. A random number generator based on the combination of four LCGs. *Mathematics of Computer Simulation* 44(1):99-107.

Lin, Y. B. 1992. Parallelism analyzers for parallel discrete event simulation. *Transactions on Modeling and Computer Simulation (TOMACS)* 2(3).

Misra, J. 1986. Distributed discrete event simulation. *Proceedings of the ACM Computing Survey* 18:39-65.

Nicol, D., and J. Liu. 2002. Composite synchronization for parallel distributed event simulation. *IEEE Transactions on Parallel and Distributed Systems* 13(5).

Nicol, D. M. 1988. Parallel discrete-event simulation of FCFS stochastic queueing networks. *SIGPLAN Notice* 23:124-137.

Prasad, S. K., and B. Naquib. 1995. Effectiveness of global event queues in rollback reduction and dynamic load balancing in optimistic discrete event simulation. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*. 187-190. Lake Placid, NY.

Prasad, S. K., S. Sawant, B. Naqib, and D. Harsch. 1994. Performance of parallel heap com- pared with parallelized calendar queue and con- current heap on a shared memory computer. Technical Report Jan-9-94-1, Department of Mathematics and Computer Science, Georgia State University, Atlanta, GA.

Preiss, B. R. 1989. The Yaddes distributed discrete event simulation specification language and execution environments. In *Proceedings of the SCS Multiconference on Distributed Simulation*, Volume 21, 139–144.

Reynolds, P. F., Jr. 1988. A spectrum of options for parallel simulation. In *Proceedings of the 1988 Winter Simulation Conference*, 325-332.

Riley, G. F. 2003. Large-scale network simulations with GTNetS. *In Proceedings of the 2003 Winter Simulation Conference*, 676-684.

Srinivasan, S., and P. F. Reynolds. 1998. Elastic time. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*.

Szymanski, B., Y. Liu, and R. Gupta. 2003. Parallel network simulation under distributed Genesis. In *Proceedings of the 17th Workshop on Parallel and Distributed Simulation*, 61.

Uspensky, J. V. 1937. *Introduction to mathematical probability* (page 18). McGraw-Hill.

Wikipedia. 2006. HyperTransport. `<http://en.wikipedia.org/wiki/Hypertransport>`.

Zhou, S., W. Cai, S. J. Turner, and F. Lee. 2002. Critical causality in distributed virtual environments. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation (PADS '02)*, 53–59.

Zimmerman, D. M., and M. Chandy. 2005. A parallel algorithm for correlating event streams. In *Proceedings of the International Parallel and Distributed Processing Symposium*.

AUTHOR BIOGRAPHIES

DAVID W. BAUER is a Senior Simulation Systems Engineer at the MITRE Corporation. He received the Ph.D., M.S., and B.S. from Rensselaer Polytechnic Institute in 2005, 2004, and 2000, respectively. Prior to joining MITRE, he was a research scientist at AT&T and GE. His research interests include parallel and distributed systems, simulation, wired and wireless networking, and computer architecture. His e-mail address is `<dwbauer@mitre.org>`.

CHRISTOPHER D. CAROTHERS is an Associate Professor in the Computer Science Department at Rensselaer Polytechnic Institute. He received the Ph.D., M.S., and B.S. from Georgia Institute of Technology in 1997, 1996, and 1991, respectively. Prior to joining RPI, he was a research scientist at the Georgia Institute of Technology. His research interests include parallel and distributed systems, simulation, networking, and computer architecture. His e-mail address is `<chrisc@cs.rpi.edu>`.