Performance Prediction using Simulation of Large-scale Interconnection Networks in POSE

Terry L. Wilmarth, Gengbin Zheng, Eric J. Bohm, Yogesh Mehta,
Nilesh Choudhury, Praveen Jagadishprasad and Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
{*wilmarth, gzheng, ebohm, ymehta, nchoudh2, jagadish, kale*}*@cs.uiuc.edu*

Abstract

Parallel discrete event simulation (PDES) of models with fine-grained computation remains a challenging problem. We explore the usage of POSE*, our Parallel Object-oriented Simulation Environment, for application performance prediction on large parallel machines such as BlueGene. This study involves the simulation of communication at the packet level through a detailed network model. This presents an extremely fine-grained simulation: events correspond to the transmission and receipt of packets. Computation is minimal, communication dominates, and strong dependencies between events result in a low degree of parallelism. There is limited look-ahead capability since the outcome of many events is determined by the application whose performance the simulation is predicting. Thus conservative synchronization approaches are challenging for this type of problem. We present recent experiences and performance results for our network simulator and illustrate the utility of our simulator through prediction and validation studies for a molecular dynamics application.*[1]

1. Introduction

PDES applications with fine-grained computation present a significant challenge. We present our simulation environment, POSE, in which we have studied the major obstacles to effective parallelization of discrete event models. We have focused our studies on models with fine computation granularity and a low degree of parallelism. POSE addresses these difficulties with a flexible object model based on the concept of virtualization, adaptive synchronization strategies, communication optimizations and load balancing. We discuss POSE in Section 2.

Simulating communication at the packet-level of a real application on a detailed contention-based network model for a large parallel computer is a good example of a discrete event simulation that is difficult to parallelize. This application presents us with all the challenges we would like POSE to overcome. It is also an area of immediate interest. New parallel computers with tremendous computational power are being constructed with surprising frequency. One example is the IBM BlueGene/L which will have 128K processors and achieve 360 TeraFLOPS peak performance. Developing and/or porting applications for/to such machines will present new challenges in getting applications to scale to such large machines. To make use of such machines efficiently, it would help to know how target applications might perform on them. If we can simulate how the application would behave on the large machine, we might be able to improve the design of the machine before it is built. We could also predict and optimize the performance of the application so that it is ready as soon as the machine is available. Another motivation for this approach is that even for existing machines, execution time for large applications is costly and often involves considerable waiting periods. A simulator could be used to prepare the application by enabling the programmer to perform debugging and optimization on the application before running on the target machine.

The BigSim[29, 30] project incorporates several levels of simulation to accurately predict performance of parallel applications on specific large parallel machines. This paper focuses on simulation of detailed contention-based network models for large parallel machines. It is impossible to use sequential simulation for large appli-

1 This work was supported in part by the National Science Foundation (NGS 0103645), the National Institutes of Health (PHS 5 P41 RR05969-04), the Defense Advanced Research Project Agency (NBCH30390004) and the Department of Energy (B341494).

cations due to the size of the data involved. As we shall see, parallel simulation of networks with common network traffic patterns is difficult, while simulating the network with traffic generated by a real application is even more challenging. The dependencies that exist between events are strong and can result in an availability of work on only a portion of the network at a time, in turn resulting in a low degree of parallelism. Further, each individual message results in a long critical path of events. In addition, since application computation is performed in an earlier stage and recorded in log files, we are only modeling packet transmissions in postmortem simulation. Thus there is very little actual computation, resulting in an extremely small grainsize. Finally, execution of a log-based simulation makes it difficult to use look-ahead to determine the safety of an event. This limits us to the use of optimistic synchronization. We discuss BigSim further in Section 3.

We present our recent experiences with network simulation and show performance results of our detailed network simulator in Section 4. We illustrate the utility of our simulator through validation and prediction studies for a molecular dynamics application in Section 5. Finally we discuss our future plans for POSE and BigSim and present our conclusions in Section 6.

1.1. Related work

As an optimistically-synchronized PDES environment, POSE is based on the Time Warp[10, 7] mechanism. POSE uses a variety of adaptive[6] protocols. One of the throttling mechanisms used by POSE is the time barrier described in MTW[22]. Breathing Time Warp[24] followed a similar approach. POSE adapts the size of the time window (similar to [5]) according to simulation behavior. A new throttling mechanism used by POSE uses an adaptable tolerance which caps the quantity of *speculative* events that can be executed based on the concept of *useful work*[17]. This approach is particularly effective at dealing with bursts of events at the same timestamp. POSE uses a variant of *speculative execution*[16] which was originally developed to make conservative protocols more aggressive. POSE uses speculation to increase aggressiveness to compensate for the throttling mechanisms used and to complement the virtualization-based object model. The approach is lighter-weight but riskier than Breathing Time Buckets[23] and Breathing Time Warp because it allows speculative events to generate future events normally.

SMART [18] is a sequential simulator for simulating diverse traffic patterns for some common network architectures. The sequential simulator would limit the scalability of the network and traffic. Another serial simulation of IBM SP systems was [26]. A conservative parallel simulation of IBM SP2 network was shown in [3]. MINSimulate [25] focuses on multistage interconnection, limiting it to a subset of current network topologies. [2] and [13] focus their efforts on K-ary N-cube interconnection networks.

Program simulation is used to study the performance of complex applications when analytical performance prediction is impractical. It is well known that simulations of such large systems tend to be slow. The direct-execution [21, 1, 20] approach uses available hardware resources to directly execute application code to simulate architectural features of interest. Á la carte[4] is a Los Alamos computer architecture toolkit for extreme-scale architecture simulation which uses a conservative synchronization engine, the Dartmouth Scalable Simulation Framework (DaSSF)[14]. They have also targeted their simulations to thousands of processors. LAPSE[8] is another system which simulated message-passing programs. However, most existing simulators are designed for MPI applications. They do not readily apply to the broader class of parallel languages such as message-driven paradigms like CHARM++. In message-driven parallel applications, computation is invoked when a message arrives. Computation order depends on message arrival order. In direct execution, messages are not delivered in the simulating environment in the same order as they would be on the target machine. This behavior of parallel execution complicates simulation further.

2. POSE

POSE is a general-purpose optimistically-synchronized PDES environment designed for simulation models with fine computation granularity and a low degree of parallelism. POSE[27] is implemented in CHARM++[12], a C++-based parallel programming system that supports the *virtualization* programming model. Virtualization involves the decomposition of a problem into N entities that will execute on P processors[11]. N is independent of P, though ideally $N >> P$. The application programmer's view of the program is of these entities and their interactions; the underlying run-time system keeps track of the mapping of entities to processors and handles any remapping that might be necessary at run-time.

In CHARM++, these entities are known as *chares*. Chares are C++ objects with special *entry* methods that are invoked asynchronously from other chares. Since many chares are mapped to a single processor, CHARM++ uses *message-driven execution* to determine which chare gets control on a proces-

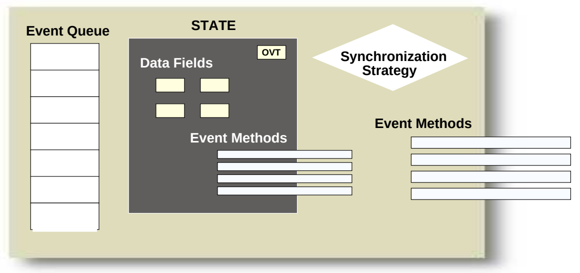

Figure 1: Components of a poser

sor. A dynamic scheduler runs on each processor and has a list of *messages* (entry method invocations) to select from. The messages are sorted according to a queuing strategy (FIFO by default). The user can attach priorities to messages. The scheduler takes the next message from its queue and invokes the corresponding entry method on the appropriate object. One 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 on the same processor that can run in the interim. Thus, using virtualization allows us to maximize the degree of parallelism. The logical processors (LPs) from PDES (called *posers* in POSE) are implemented with CHARM++'s chares. We use timestamps on messages as priorities and the CHARM++ scheduler serves as a presorting event queue.

2.1. The POSE object model

Posers represent sequential entities in the simulation model. They are similar to LPs, but should encapsulate much smaller portions of state. POSE requires that parallelism in the model be represented by posers in the simulation handling events in parallel, but the responsibility for decomposing a simulation model into the smallest components possible (maximizing N, the number of posers) lies with the programmer. This approach differs from timelines[15] in DaSSF and similar approaches in that it encourages the maximum amount of decomposition without regard for how objects might relate. As we shall see, this choice does not result in significant drawbacks.

Figure 1 illustrates the structure of a poser. Each poser has an *object virtual time* (OVT). This is the virtual time that has passed since the start of the simulation relative to the object. Posers have *event methods* which are CHARM++ *entry methods* receiving *timestamped* messages. The poser encapsulates the state specified by the programmer and contains a local event queue. Incoming events are redirected from the CHARM++ scheduler into the appropriate poser's local event queue.

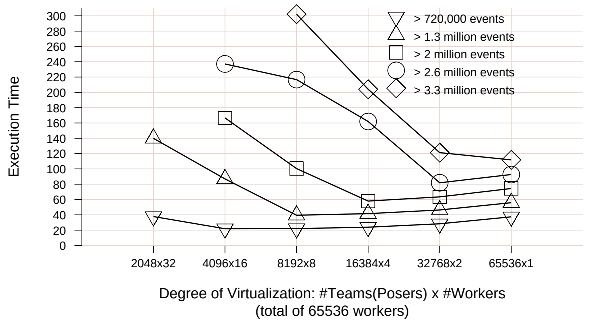

Figure 2: Effects of virtualization

Events are processed according to an instance of a synchronization strategy associated with the poser.

In addressing the problem of fine-granularity, this approach seems counterintuitive to approaches that cluster entities together. However, decomposing a model into posers has many benefits. The poser event queue limits the scope of simulation activity to the poser itself; since different entities may have dramatically different behaviors, this limits the effects of those behaviors to a smaller scope. When checkpointing is performed periodically according to the number of events received by a poser, this structure enables less frequent checkpointing on smaller states, reducing overhead and memory usage. Rollbacks are less likely to occur when the scope affected by incoming events is smaller. When they do occur, they cause shorter rollbacks. Decomposing state into smaller objects makes LP migration simpler. Finally, this structure paves the way for adaptive synchronization strategies that can fine-tune each LP's behavior.

The 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, we need to examine the tradeoffs. We ran a set of experiments varying the degree of virtualization of a fixed-size problem using a basic optimistic strategy with no throttling mechanisms. 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 Cool[2] with a synthetic benchmark that allows parallel *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 organizes the workers in rings such that 50% of the

2 Cool is a Linux cluster of 8 quad Xeon SMPs with fast Ethernet.

Table 1: Overhead cost breakdown

	Teams × Workers per Team				
	4096×16	8192×8	16384×4	32768×2	65536×1
FE	38.18	36.19	31.15	31.64	32.33
GVT	61.22	11.26	1.09	1.11	1.27
Sync	1.35	2.37	1.08	1.24	1.42
CP	3.96	3.45	0.98	0.78	0.55
FC	15.40	12.70	1.78	1.89	2.00
RB	0.00	0.00	0.00	0.00	0.00
Com	14.57	14.00	13.98	14.42	15.62
Other	16.28	7.90	6.10	10.90	19.20
Total	149.61	87.87	56.16	61.98	72.39

workers communicate locally and the rest remotely. We simulated a total of 65,536 workers in each experiment 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 Cool with several different program sizes (in terms of number of events handled). The results are shown in Figure 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 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. Missing points for a curve indicate that those degrees of virtualization ran out of either memory or time allotted for the run.

We show a breakdown of the performance in Table 1. This table shows the costs in s (averaged per processor) for various overhead types for the >2 million event problem size, varying the degree of virtualization. FE is forward execution, CP is checkpointing, FC is fossil collection and RB is rollback and cancellation. The optimal performance for this set of runs occurs with 16,384 teams of 4 workers each. Activities which are highly dependent on the quantity of objects in the simulation (GVT, synchronization, 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 creating, distributing and managing the objects in the CHARM++ runtime system. Checkpointing costs decrease consistently as degree of virtualization increases. The most significant performance differences are attributable to higher memory usage for lower degrees of virtualization. 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 occurs at the lowest degree of virtualization. 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 2 does not show is that a higher degree of virtualization allows us to run a program on more processors than we could with lower degrees of virtualization, further increasing the chances of obtaining a better speedup. Thus, this "fine granularity of entities" does not incur significant additional cost and has further benefits that we are not yet taking advantage of. For example, having more fine-grained entities is a great benefit when we use load balancing in our simulations. As we shall see in the next section, high degrees of virtualization will enable fine-tuned adaptive synchronization.

2.2. Adaptive synchronization

In POSE, an object gets control of a processor when it either receives an event or cancellation message via the scheduler, or when a new GVT estimate has been calculated. In the first case, the object's synchronization strategy is invoked. In the second case, we perform fossil collection before invoking the strategy. Our basic optimistic strategy handles cancellation messages first and then checks for any stragglers that may have arrived and rolls back to the earliest. Finally, it is ready to perform forward execution steps.

This is where the opportunity to perform *speculative computation* arises. In traditional optimistic approaches, an event arrives, is sorted into the event list and the earliest event is executed. The event is the earliest on the processor, but may not be the earliest in the simulation, so its execution is speculative. In our approach, we have a *speculative window* that governs how far into the future beyond the GVT estimate an object may proceed. Speculative windows are similar to the time windows[22] used to throttle optimism in other systems. They differ in how events within the window are processed. In POSE, *all* events on a poser with timestamp within the speculative window are executed as a *multi-event*. 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. The strategy speculates that those events are the earliest that the *poser* will receive. This approach is more aggressive than traditional approaches because our version of speculative execution does not "hold back" events that are spawned, to be sent later when we are sure it is "safe". Rather they are executed in the standard way; spawned events are actually sent to other, possibly remote, posers. Multi-events are discussed in detail in [28].

Using multi-events, we reduce scheduling and context switching overhead and benefit from a warmed cache, eliminating some of the drawbacks associated with the high degree of virtualization. This form of speculation may increase rollback overhead. However, since the speculative window itself is the throttling mechanism, and since rollback costs are reduced by being highly localized due to the high degree of virtualization, we have many ways to keep rollback cost under control. We now combine the concepts of poser virtualization and speculation with adaptive synchronization.

In POSE, 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).

POSE allows for any strategy to be used for posers of a particular type, and many strategies have been written. Our ultimate goal however has been to develop a single default strategy flexible enough to adapt to a variety of situations. For this paper, we describe the behavior of this default strategy called *Adept*.

Adept applies three throttling mechanisms to the aggressive speculation behavior. First, the speculative window size can be adapted. This controls how far into the future a poser can progress. Initially, the window size is infinite. When a rollback occurs, the window is reduced to the interval between the GVT estimate and the time at which the straggler arrived. Assuming no other stragglers arrive, the window is rapidly expanded to infinity again after the poser has control only a few times.

The second throttling mechanism limits the *quantity* of events comprising a multi-event. A *speculative tolerance* is a percentage of total events executed that did not prove useful. If a poser is under the tolerance, it may execute however many events it can that would keep it under the tolerance, assuming none of the events turned out to be useful. A non-zero minimum guarantees progress.

The third throttling mechanism takes a snapshot of the memory footprint of an object, and if it exceeds a maximum, does not allow the object to process more events until fossil collection has been performed. Again, if the object has work that would restrict GVT advancement, it is executed to guarantee progress.

The last two throttling mechanisms are adapted to each poser. However, the criteria used to determine when to throttle is collected at processor level, so there is additional flexibility in how objects can behave. For example, if some objects are using very little memory, others which need more can be allowed to use it.

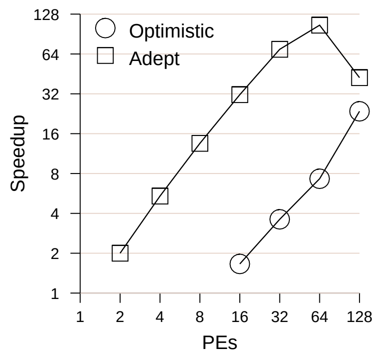

Figure 3: Adept vs. Optimistic

2.3. POSE performance

Early experiments with a simple 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 show a Turing[3] run of Adept vs. Optimistic in Figure 3 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. Optimistic was unable to complete runs 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.

As our default strategy for POSE, Adept exhibits excellent scalability. Figure 4 shows how POSE using Adept scales with problem size using the Multiple Rings benchmark. In one experiment, we double the number of events handled per object. Doubling processors then

3 *Turing* is a cluster of 640 dual 2GHz G5 processors 4 GB RAM Apple Xserves connected by Myrinet.

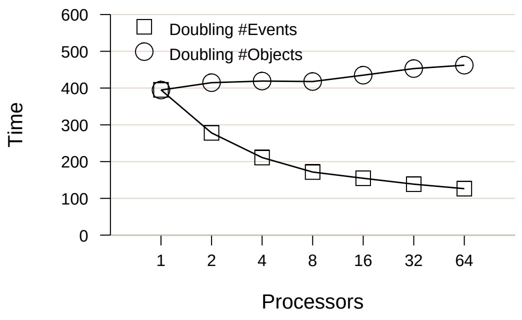

Figure 4: Scaling problem size

results in a decrease in the time taken to execute the double-size simulation. This is largely due to the increase in the size of the multi-events. More events can be handled in less time since there is less synchronization overhead and improved cache performance. Doubling problem size by doubling the number of objects has quite a different result and clearly illustrates the cost of increasing the degree of virtualization. The time gradually increases as processors and objects double. This is largely due to the startup time taken to construct the additional objects on the additional processors, but also includes extra overhead for object management. Since the number of events per objects remains constant, the size of multi-events cannot increase.

3. Network simulation

The BigSim project aims at developing techniques and methods to facilitate the development of efficient scalable applications on very large parallel machines. BigSim is based on simulation and performance prediction techniques including network simulation.

Our work in this project has focused on two distinct user groups. For the scientific community we have produced a simulation environment[31, 30] for performance prediction of scientific applications running on large machines. For designers of HPC systems we have created a detailed simulation environment for the performance prediction of future high speed large scale interconnects.

3.1. Application performance prediction

The core of the application performance prediction system is the BigSim emulator [29, 31]. Using the virtualization of CHARM++ [11], the BigSim emulator presents the execution environment of a petaflops class machine. CHARM++ or AMPI[9] applications are compiled to run on the emulator just as though it were any other architecture. They can then be run on an existing parallel computer and their performance captured in an

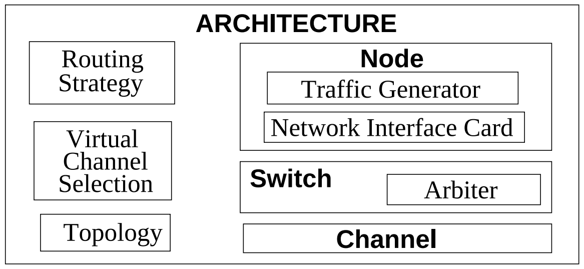

Figure 5: BigNetSim conceptual model

event trace log file. The trace log can then be used for performance analysis. The network simulation included within BigSim is a simple latency based network model.

When more precise network behaviors must be studied, the trace can be loaded into the POSE-based BigNetSim which corrects the recorded timestamps based on network contention, adaptive routing, hybrid networks, etc. as determined by the particular model chosen.

3.2. Network performance prediction

BigNetSim facilitates informed decision making for 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 (see Figure 5). BigNetSim currently includes support for Torus, 3D-Mesh, Fat-Tree and Hypercube topologies.

In Figure 5, the traffic generator, NIC, Switch and Channel are modeled as posers. They rely exclusively on event messages to communicate with other simulation objects. Messages created on a node are passed to the NIC, which packetizes the message and sends the packets to a Channel. The Channel forwards a packet to a Switch. The Switch has an Arbiter, which uses the configured routing logic, to select the output Channel to route the packet. Switch models Virtual Channels and flow control based on feedback from downstream Switches and VC selection strategies. Channels in the path to the destination node receive the packets and pass them up through Switches. Finally the last channel in the path passes it to the NIC, which reassembles packets to form a message and sends it to the node. Several network parameters such as packet-size, channel-bandwidth, channel-delay, switch-buffer-size and a host of NIC delay parameters can be set in a network configuration file, which the simulation reads at runtime.

BigNetSim's modular nature and the detail of model entities should enable accurate modeling of current and future interconnection networks with minimal additional

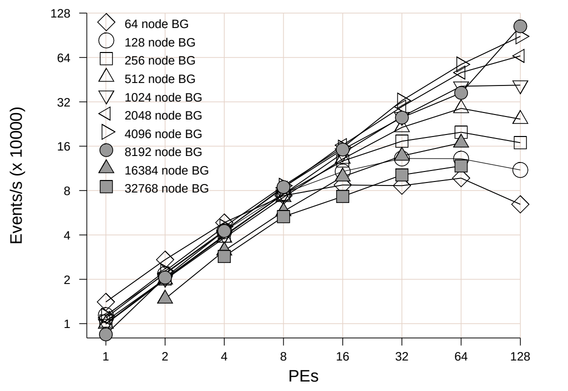

Figure 6: BigNetSim events/sec with TrafficGen

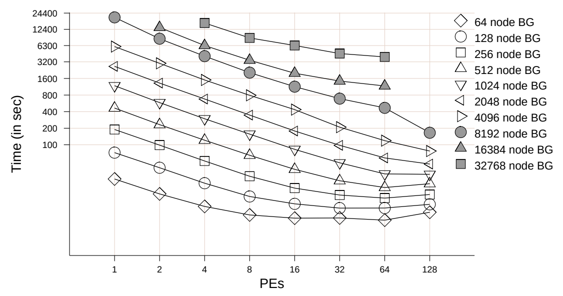

Figure 7: BigNetSim time with TrafficGen

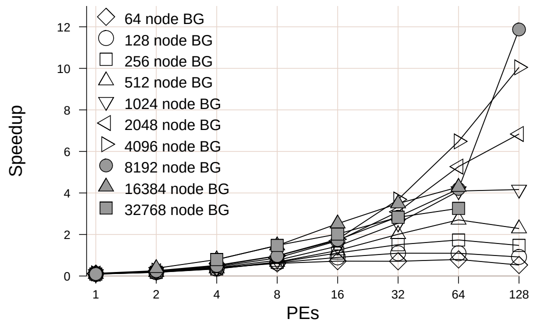

Figure 8: BigNetSim speedup with TrafficGen

code. Higher resolution modeling of NIC behavior is planned. No modeling is done of message handling processes, such as message protocol stack manipulations.

When evaluating a network, it is essential to be able to study network performance based on expected traffic loads. The user may choose to load the network by executing an application trace or create standard network load patterns using the TrafficGen interface. TrafficGen sends messages from a source node to a destination node chosen according to a predefined pattern (k-shift, ring, bit transpose, bit reversal, bit complement and uniform distribution). These patterns are based on communication patterns in specific applications. Messages are generated by a deterministic or Poisson distribution.

As shown in Section 4, POSE's support for virtualization enables BigNetSim to perform and scale well despite the seemingly heavyweight object implementation.

4. Performance

We have evaluated the performance of our network simulator by simulating a uniform distribution of random traffic using our TrafficGen module on a simulated 3D Torus network ranging in size from 4x4x4 to 64x32x32. This pattern was chosen because it would more closely resemble the behavior of an arbitrary collection of applications running on a supercomputer than the other standard patterns. The randomly chosen destinations result in a repeatable but random asymmetric load on the network. The network parameters were selected to simulate the direct torus network of the BlueGene/L machine. All runs were made on Turing.

Figure 6 shows the number of POSE-committed events per second on average. From a network simulation perspective, a single packet-hop in a network translates to 6 or 7 events on an average. The system scales well up to 64 processors, with larger networks exhibiting the best scaling beyond 64 processors. At the peak we achieve 1 million committed events (150,000 packet-hops) per second on 128 processors for the 8192-node network. We observe that with increasing number of posers per processor, the number of events/s decreases. Another observation is that the number of events/s seems to scale well with increasing number of processors regardless of the problem size or the number of posers.

Figure 7 shows the execution time of BigNetSim for various network sizes. In this plot, the amount of work per poser is constant, but the number of posers/processor doubles with doubling network size. Hence, the time doubles too. BigNetSim shows good scalability upto a point beyond which it saturates. This can be explained by the tradeoff between virtualization and the costs associated with increasing the number of processors as expected from our discussion in Section 2. The point of saturation is around 16 processors for smaller problem sizes (simulation of a 4x4x4 network) and it increases with the problem size. Thus, BigNetSim scales better with increasing problem size. If we were to look at self-speedup (speedup relative to 1 processor parallel simulation), we observe linear speedup (evident from Figure 7).

Figure 8 shows a plot of speedup relative to sequential simulation time. The distinction between sequential and parallel simulation should be well understood. Sequential simulation orders events and executes them without communication or synchronization overhead.

Parallel simulation has to execute events based on optimized predictions, which could cause out of order execution and hence rollbacks. Therefore, parallel simulation on one processor is nearly 10 to 12 time slower than sequential simulation, but it has the promise that with good scaling on a large number of processors it can perform much better than sequential simulation. Furthermore, parallel simulation is essential for problems which are too large for a single machine. So, we look for a small break-even point and good scaling above that. Our simulation demonstrates both these features. The average breakeven point is 8 or 16 processors and it scales well beyond that. A promising way to decrease the discrepancy between sequential and parallel execution on one processor is by further decomposing the posers into a greater quantity of smaller objects, thus, increasing the degree of virtualization. This is expected to improve performance as shown in the virtualization benchmark of Section 2.3.

In the three figures, there are certain missing points. These are for large number of posers on a small number of processors which causes it to run out of memory. This strenghtens the need for a parallel simulation. For a network size above 8192 nodes, a single processor run (whether sequential or parallel) runs out of memory. In the speedup plot we have estimated values for these missing points based on available sequential data for smaller network sizes.

5. Performance prediction and validation

We have evaluated BigNetSim for real applications on very large numbers of processors. One such application is the simulation of biomolecules using Molecular Dynamics. It is one of the important applications for BlueGene/L and other large parallel machines. The application we chose is NAMD [19], which is the current state-of-art high performance molecular dynamics code designed for large biomolecular systems. It is one of the recipients of 2002 Gordon Bell Award for the achievement of scaling parallel simulation to 3000 processors.

5.1. NAMD validation

We have compared the actual execution time of NAMD with our simulation of it using BigSim on LeMieux[4]. Our validation approach is as follows. We first run NAMD on a number of real processors and record the actual run time. Then we run NAMD on BigSim emulator with a much smaller number of processors simulating the same number of processors used in the original run. This emulation generates log files that we then simulate with BigNetSim running the simple latency-based network model. We record the run time predicted by the simulator and compare with the original run time.

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. The result is shown in Table 2. The first row shows the actual execution time per timestep of NAMD on 128 to 2250 processors on LeMieux. The second row shows the predicted execution time per timestep using BigNetSim on a Linux cluster with network parameters based on Quadrics network specifications. It shows that the simulated execution time is reasonably close to the actual execution time. The predicted run time is not as accurate when number of real processors grows larger. One possible reason is that we might not have modeled the cache performance and the memory footprint effects in enough detail.

Table 2: Actual vs. predicted time (in *ms*) per timestep for NAMD

PEs	128	256	512	1024	2250
Actual time	71.5	40.3	23.9	17.6	12.8
Predicted time	75.8	43.6	25.1	20.8	16.13

5.2. Parallel performance

To evaluate the parallel performance of the simulator itself, we ran the BigSim emulator on 64 real processors with the NAMD program on 2048 simulated nodes configured in a 3D Torus topology with Bluegene/L characteristics. The emulation traces were then used with BigNetSim on a varying number of processors.

We show simulation execution time for BigNetSim with NAMD from 2 to 128 processors in Figure 9 and a corresponding speedup plot relative to one processor parallel time in Figure 10. It shows that BigNetSim scales linearly to 8 processors and sub-linearly to 26 on 32 processors. The simulator continues to scale to 63 on 128 processors.

The results in Figure 10 show that BigNetSim has decent parallel performance even when applied to performance prediction of a real world application. This is much more challenging than a synthetic benchmark due to complex dependencies among events resulting in relatively limited parallelism in the simulation. Performance

4 LeMieux is 750 Quad Alphaserver ES45 node machine at Pittsburgh Supercomputing Center (PSC)

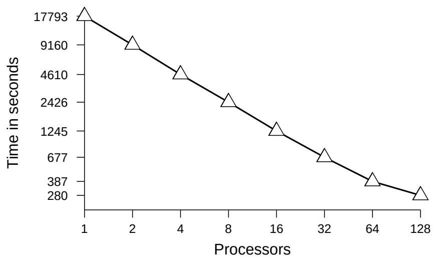

Figure 9: BigNetSim execution time with NAMD

is further complicated by the addition of several objects for each simulated node to handle trace input and perform message delivery confirmation.

This particular simulation achieved breakeven with the sequential simulator at 128 processors. BigNetSim was able to achieve a much earlier breakeven (16 processors, see Figure 8) for the 2048 node network when using TrafficGen. The reduced parallelism of the application trace revealed some implementation problems in BigNetSim which had been masked by the greater parallelism of artificial network loads. On closer inspection of the execution statistics provided by POSE, we found that BigNetSim execution time was dominated by rollback and GVT overhead. Closer analysis of the BigNetSim implementation indicates that the Switch object is too large and should be decomposed into smaller objects. Smaller objects take advantage of POSE performance characteristics as described in Section 2.

Simulation of detailed contention-based network models for predicting parallel performance is still quite challenging, but we have also achieved decent speedups relative to one-processor parallel time in [30]. An additional challenge is that the Apo-Lipoprotein A1 benchmark itself does not scale well beyond 3000 processors. This meant we could not get a credible result for the large network sizes (i.e. 8192) where BigNetSim achieves its best performance. The 2048 node run resulted in a small problem size of 3.5 million events, resolved by the sequential version of BigNetSim in a few minutes on a single processor.

6. Conclusions and future research

We have seen that simulation of interconnection networks for very large parallel computers presents a significant challenge, particularly when we wish to simulate their behavior for performance prediction of real applications. This problem presents two of the major drawbacks that make PDES challenging: fine granularity of computation and low degree of parallelism.

POSE has made it possible to study this and other problems by improving the chances of being able to scale such problems to greater numbers of processors. It

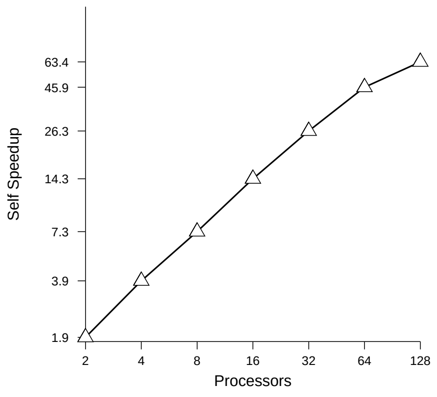

Figure 10: BigNetSim speedup with NAMD

incorporates a new object model based on virtualization and adaptive speculative strategies that can tune simulation behavior to a variety of applications. For our TrafficGen detailed network simulations, BigNetSim achieves excellent problem-size scaling, particularly for larger problem instances on greater numbers of processors. BigNetSim also exhibits excellent self-scaling (relative to single processor parallel runs). It is also capable of achieving break-even points relative to sequential time at 8 processors.

In the future, we plan to explore and expand the POSE load balancing framework as that seems key to improving the performance of this particularly challenging application. We also plan to enhance the adaptive capabilities of our synchronization strategies. BigNetSim's oversized Switch object will be broken up into smaller objects to lower the breakeven point and increase overall performance. Furthermore, higher fidelity modeling, specifically NIC performance constraints, will be studied. Performance prediction for other applications (such as Finite Element Simulations, larger NAMD benchmarks and Car Parinello Ab Initio Molecular Dynamics) will be undertaken. We anticipate that imminent access to actual Bluegene/L machines will allow us to improve the overall fidelity of the BigSim Project.

References

[1] V. S. Adve, R. Bagrodia, E. Deelman, and R. Sakellariou. Compiler-optimized simulation of large-scale applications on high performance architectures. *Journal of Parallel and Distributed Computing*, 62:393–426, 2002.

[2] A. Agarwal. Limits on Interconnection Network Performance. In *IEEE Transactions on Parallel and Distributed Systems*, October 1991.

[3] C. Benveniste and P. Heidelberger. Parallel simulation of the IBM SP2 interconnection network. In *Winter Simulation Conference*, 1995.

[4] K. Berkbigler, G. Booker, B. Bush, K. Davis, and N. Moss. Simulating the Quadrics Interconnection

Network. In *High Performance Computing Symposium 2003, Advance Simulation Technologies Conference 2003*, Orlando, Florida, April 2003.

[5] S. R. Das. Estimating the cost of throttled execution in time warp. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pages 186–189, May 1996.

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

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

[8] P. M. Dickens, P. Heidelberger, and D. M. Nicol. A distributed memory lapse: parallel simulation of message-passing programs. *SIGSIM Simul. Dig.*, 24(1):32–38, 1994.

[9] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, College Station, Texas, October 2003.

[10] 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.

[11] L. 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.

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

[13] S. Kumarasamy, S. K. Gupta, and M. A. Breuer. Testing a K-Ary N-Cube Interconnection Network. In *IEEE International On-Line Testing Workshop*, June 1998.

[14] J. Liu and D. Nicol. *Dartmouth Scalable Simulation Framework User's Manual*. Dept. of Computer Science, Dartmouth College, Hanover, NH, February 2002.

[15] J. Liu, D. Nicol, B. J. Premore, and A. L. Poplawski. Performance prediction of a parallel simulator. In *PADS '99: Proceedings of the thirteenth workshop on Parallel and distributed simulation*, pages 156–164. IEEE Computer Society, 1999.

[16] H. Mehl. Speedup of conservative distributed discrete-event simulation methods by speculative computing. In *Proceedings of the Multiconference on Advances in Parallel and Distributed Simulation*, pages 163–166, January 1991.

[17] A. C. Palaniswamy and P. A. Wilsey. Adaptive bounded time windows in an optimistically synchronized simulator. In *Great Lakes VLSI Conference*, pages 114–118, March 1993.

[18] F. Petrini and M. Vanneschi. SMART: a Simulator of Massive ARchitectures and Topologies. In *International Conference on Parallel and Distributed Systems Euro-PDS'97*, Barcelona, Spain.

[19] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD.

[20] S. Prakash and R. L. Bagrodia. Mpi-sim: Using parallel simulation to evaluate mpi programs. In *Proceedings of IEEE Winter Simulation Conference*, 1998.

[21] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The wisconsin wind tunnel: Virtual prototyping of parallel computers. In *Measurement and Modeling of Computer Systems*, pages 48–60, 1993.

[22] L. M. Sokol, J. B. Weissman, and P. A. Mutchler. Mtw: an empirical performance study. In *Winter Simulation Conference*, pages 557–563. IEEE Computer Society, 1991.

[23] J. Steinman. SPEEDES: Synchronous Parallel Environment for Emulation and Discrete Event Simulation. In *Proceedings of the 1991 SCS Multiconference on Advances in Parallel and Distributed Simulation*, pages 95–103, January 1991.

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

[25] D. Tutsch and M. Brenner. MINSimulate: A Multistage Interconnection Network Simulator. In *17th European Simulation Multiconference: Foundations for Successful Modeling and Simulation*, 2003.

[26] M. Uysal, A. Acharya, R. Bennett, and J. Saltz. A Customizable Simulator for Workstation Networks. In *11th International Parallel Processing Symposium (IPPS '97)*, Geneva, Switzerland.

[27] T. Wilmarth and L. V. Kalé. Pose: Getting over grainsize in parallel discrete event simulation. In *2004 International Conference on Parallel Processing*, pages 12–19, August 2004.

[28] T. L. Wilmarth. *POSE: Scalable General-purpose Parallel Discrete Event Simulation*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[29] G. Zheng, G. Kakulapati, and L. 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.

[30] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. V. Kalé. Simulation-based performance prediction for large parallel machines. In *International Journal of Parallel Programming*, number to appear, 2005.

[31] G. Zheng, T. Wilmarth, O. S. Lawlor, L. V. Kalé, S. Adve, D. Padua, and P. 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.