BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines

Gengbin Zheng, Gunavardhan Kakulapati, Laxmikant V. Kalé
Dept. of Computer Science
University of Illinois at Urbana-Champaign
{gzheng, kakulapa, kale}@cs.uiuc.edu

Abstract

We present a parallel simulator — BigSim — for predicting performance of machines with a very large number of processors. The simulator provides the ability to make performance predictions for machines such as BlueGene/L, based on actual execution of real applications. We present this capability using case-studies of some application benchmarks. Such a simulator is useful to evaluate the performance of specific applications on such machines even before they are built. A sequential simulator may be too slow or infeasible. However, a parallel simulator faces problems of causality violations. We describe our scheme based on ideas from parallel discrete event simulation and utilize inherent determinacy of many parallel applications. We also explore techniques for optimizing such parallel simulations of machines with large number of processors on existing machines with fewer number of processors. [1]

1 Introduction

Parallel machines with an extremely large number of processors are now being designed. For example, the BlueGene (BG/L) machine [3] being designed by IBM will have 64,000 dual-processor nodes. More near term large machines include *ASCI Purple* [2] with about 12k processors and *ASCI Red Storm* with 10k processors. A design from IBM, code-named Cyclops (BlueGene/C), had over one million floating point units, fed by 8 million instructions streams supported by individual thread units. In response to an initiative by the U.S. Department of Defense, three vendors are designing even more adventurous architectures for machines to be built within the next 5-7 years.

In this context, it is essential to be able to evaluate the performance of planned machines before they are built. However, parallel performance is notoriously difficult to model. It is clearly not adequate to multiply the peak floating point performance of individual processors by the number of processors. Performance of communication subsystems, complex characteristics of target applications during their multiple phases and the behavior of runtime support systems, interact in nonlinear fashion to determine overall performance. For example, for a particular problem size, communication latencies may be completely masked by effective overlap of communication and computation by the application. However for the same application, if the CPU speed is increased above a threshold, the performance may be totally dominated by the communication performance.

Even for existing large machines, our performance prediction approach is useful. For example, consider the process of performance optimization of a complex application. Time on large machines is hard to get and must be reserved well ahead of time. A performance measurement run typically takes only a few minutes. However, every time such a run is carried out, application developers must spend some time visualizing and analyzing the performance data before the next set of optimizations are decided upon, at which point one must wait in queue for the next running slot (typically at least for a day). With a simulator, this performance debugging cycle can be shortened considerably. [2]

We suggest that new machines be evaluated by running planned applications on full-fledged simulators of such machines. The approach we present in this paper is a step in the direction of accurate simulations of extremely large parallel machines using currently available parallel machines.

To this end, we have built a system that can emulate large parallel machines [16]. Based on the CHARM++ [10] parallel programming system, our emulator has successfully emulated several million threads (one for each target machine processor) on clusters with only hundreds of processors. However, the emulator is useful only for studying programming models and application development issues that arise in the context of such large machines. Specifically, the emulator does not provide useful performance information.

To make performance predictions, one must either (a) record traces during emulation, and then run a sequential trace-driven simulation or (b) modify the emulator to carry out a *Parallel Discrete Event Simulation* (PDES). The sequential trace-based approach is infeasible if the trace data will not fit in the memory of a single processor, or will take too long to run. The latter approach appears to be quite chal-

[1]This work was supported in part by National Science Foundation (NGS 0103645) and National Institutes of Health (PHS 5 P41 RR05969-04)

[2]Even if predictions are approximate, they are still useful to evaluate the algorithms to the first degree of approximation.

lenging, in view of the failure of typical PDES systems to provide effective parallel performance. However, we have developed a scheme, based on the inherent determinacy of parallel programs, that makes this approach feasible and efficient.

Essentially, our approach involves letting the emulated execution of the program proceed as usual, while concurrently running a parallel algorithm that corrects time-stamps of individual messages. We present an overview of our simulation system called *BigSim* in the next section. In Section 3, we introduce our approach by comparing BigSim with some related works. In Section 4, we briefly discuss the issues and techniques involved in converting the emulator to the simulator. In Section 5, we present our simulator in detail, specifically our synchronization algorithm used to reduce the simulation overhead. Finally, we present a few application benchmarks that illustrate the use of the simulator, predicting performance on a BG/L like machine with 64,000 processors.

2 BigSim Overview

In order to simulate a future massive parallel machine on an existing parallel machine with only hundreds of processors, one physical processor has to simulate hundreds or even thousands of processors of the target machine. The total memory requirement for the simulation may raise a red flag. However, simulation is still possible because:

- Some planned machines have low memory-to-processor ratio such as BlueGene/C. It was originally designed to have about half a terabyte of total memory. However, each chip with 25 processors shares only 12MB of memory. Thus to emulate BlueGene/C running an application which uses the full machine will require "just" 500 processors of a traditional parallel machine with 1GB memory per processor.

- For planned machines with high memory-to-processor ratio, no existing parallel machine can have enough memory to simulate the full machine. However, many real world applications such as molecular dynamics simulation do not require a large amount of memory.

- For applications that do require a large amount of memory, it is still possible to use automatic out-of-core execution [15] to temporarily move the data in memory to disk when it is not needed immediately. This swapping increases the simulation time. However if the only thing we are interested in is the predicted running time, for a few timesteps, this is still affordable.

To simulate a class of petaflops computers, we designed a low level abstraction of such machines. In the programmer's view, each node consists of a number of hardware-supported threads with common shared memory. A runtime library call allows a thread to send a short message to a destination node. The header of each message encodes a handle function to be invoked at the destination. A designated number of threads continuously monitor the incoming buffer for arriving messages, extract them and invoke the designated handler function. We believe this low level abstraction of the petaFLOPS architectures is general enough to encompass a wide variety of parallel machines with different numbers of processors and co-processors on each node.

We have developed a software *emulator* based on this low level model. The details of the emulator and its API are presented in [16].

In this base level model, the programmer must decide which computations to run on which node. The CHARM++ runtime system built upon the emulator relieves the application programmer of the burden of deciding where the subcomputations run. Several programming languages are supported by this runtime system including MPI and CHARM++.

In order to predict the parallel performance of applications, we use the parallel discrete event simulation (PDES) methodology in BigSim. We tightly couple the BigSim PDES engines with the CHARM++ runtime system in order to improve the efficiency of the simulation. The overall architecture of BigSim is illustrated in Figure 1.

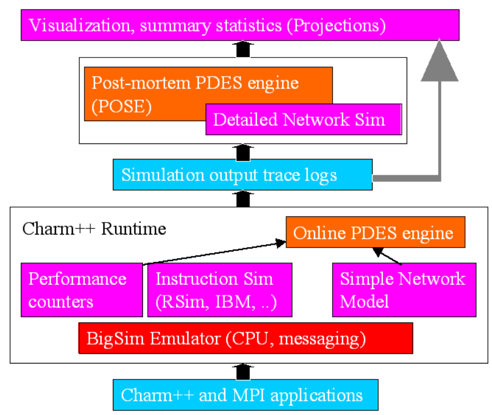


Figure 1. The architecture of BigSim

We developed two modes of PDES engines in BigSim: *online* mode and *postmortem* mode event simulation. Online mode simulation runs the parallel simulation along with the actual execution of the applications. The advantage of this *direct execution* approach is that it makes possible to simulate programs that perform runtime analysis. In this paper, we only focus on the online event simulation. As part of an ongoing project, we are also developing a postmortem network simulator that models contention-based network [20].

BigSim also provides a performance visualizer "Projections" which helps to analyze the performance characteristics of applications.

2.1 Parallel Programming Languages on BigSim

Based on the low level programming API provided by the emulator, several parallel programming languages are implemented on BigSim. They are MPI, CHARM++ [10] and Adaptive MPI [6].

MPI is a popular way to program parallel machines. However, it remains a question whether it is easy and efficient to program the next generation of supercomputers using it.

CHARM++ is an object-based portable parallel programming language that embodies message-driven execution. CHARM++ consists of parallel objects and object arrays which communicate via asynchronous method invocations. CHARM++ supports "**processor virtualization**" [8] which leads to several benefits by enabling automatic runtime optimizations. CHARM++ may be better suited for such extreme-scale machines because its adaptive runtime systems maps objects to processors thus automating resource management [19]. NAMD, a CHARM++ application, has achieved unprecedented speedup on 3000 processors on PSC Lemieux [11, 14]. Other applications such as ab Initio MD, Cosmology and Rocket Simulation have demonstrated that CHARM++ is a viable and successful parallel programming model.

Adaptive MPI [6], or AMPI, is an MPI implementation and extension based on CHARM++ that supports processor virtualization. AMPI implements virtual MPI processes (VPs), several of which may be mapped to a single physical processor. Taking advantage of CHARM++'s migratable threads, AMPI also supports adaptive load balancing by migrating AMPI threads.

In our programming environment, MPI is thus a special case for AMPI when exactly one VP is mapped to a physical processor.

In this paper, we focus on the simulations of both message passing and message driven parallel applications.

3 Related Work and Our Approach

In general, simulation for performance predictions can be carried out as *Parallel Discrete Event Simulation*(PDES), which has been extensively studied in literature [7, 5, 17].

In BigSim simulator, the simulation entities include simulated processors, network components and all software components in the user application such as processes in MPI or parallel objects in CHARM++. We map the physical target processors to logical processors(LPs), each of which has a local virtual clock that keeps track of its progress. In the simulation, user messages together with their subsequent computations play the role of events.

In the parallel simulation, each LP works on its own by selecting the earliest event available to it and processing it without knowing what happens on other LPs. Thus, methods for synchronizing the execution of events across LPs are necessary for assuring the correctness of the simulation.

Two broad categories of the strategies are *conservative* approach and *optimistic* approach.

In conservative approach, one has to ensure the safety of processing the earliest event on an LP in a global fashion. The drawback of this method is that the process of determining safety is complicated and expensive. It also reduces the potential parallelism in the program.

Optimistic synchronization protocols allows LPs to process the earliest available event with no regard to safety. When causality errors occur, a rollback mechanism is needed to undo earlier out of order execution and recreate the history of events executed on the LP as if all events were processed in the correct order of arrival. Despite the cost of synchronization, the optimistic approach exploits the parallelism of simulation better by allowing more concurrent executions of events. In BigSim, we adopt an extension of the optimistic synchronization approach.

The most well studied optimistic mechanism is Time Warp, as used in the Time Warp Operating System [7]. Time Warp was notable as it was designed to use process rollback as the primary means of synchronization.

The Georgia Tech Time Warp (GTW) [5] system was developed for small granularity simulations such as wireless networks and ATM networks. One of the GTW features that carries over into BigSim is the *simulated time barrier*, a limit on the time into the future that LPs are allowed to executed.

The Synchronous Parallel Environment for Emulation and Discrete Event Simulation (SPEEDES) [17, 18] was developed with a different optimistic approach to synchronization called *breathing time buckets*.

Á la carte [1] is a Los Alamos computer architecture toolkit for extreme-scale architecture simulation. It chose conservative synchronization engine, the Dartmouth Scalable Simulation Framework (DaSSF) [13] for the handling of discrete events. They have targeted on simulating thousands of processors. In the Quadrics network simulation of SWEEP3D [4], they reported no speedup when adding more computational nodes in the simulation. The performance data they reported in the paper is only for simulating a 36-process run of SWEEP3D using 2 to 36 real processors.

Conservative simulators require a lookahead, which imposes a high global synchronization overhead, and typically limits the amount of parallelism one can exploit in simulation. On the other hand, the optimistic general purpose simulators, when directly applied to performance prediction of parallel applications on extreme scale of processors (tens of thousands or even millions of processors), may lead to very high synchronization overhead, caused by the need to rollback a simulation when a causality violation is detected.

The synchronization overhead of optimistic concurrency control consists of:

- **Checkpointing overhead** - time spent in storing program state before an event is executed which might change that state.
- **Rollback overhead** - time spent in undoing events and sending cancellation messages.
- **Forward execution overhead** - time spent in re-executing events that were previously rolled back.

In BigSim, we found that by taking advantage of the parallel programs' inherent determinacy, the above overhead can be dramatically reduced, leading to great improvement in the simulation.

Parallel applications tend to be deterministic, with a few exceptions (such as branch-and-bound and certain classes of truly asynchronous algorithms). Parallel programs are written to be deterministic. They produce the same results, and even though the execution orders of some components may be allowed to differ in smaller time intervals, they carry out the same computations.

In order to detect and exploit the parallel programs' inherent determinacy, BigSim is designed to be integrated with language and runtime support to reduce the overhead of synchronization. In Section 5, we describe this methodology in detail for a broad class of parallel programs.

4 From BigSim Emulator to Simulator

Converting the emulator to a simulator requires correct estimation of the time taken by sequential code blocks and messaging. BigSim is capable of using various plug-in strategies for estimation of the performance of these component models.

4.1 Predicting the Time of Sequential Code

The walltime taken to run a section of code on traditional machines can not be used directly to estimate the compute time on the target machine. As we do not know the time taken for a given sequence of instructions on the target machines, we use a heuristic approach to estimate the predicted computation time on the simulator. Many possible methods are described below. They are listed in the increasing order of accuracy (and the complexity involved).

1. User supplied expression for every block of code estimating the time that it takes to run on the target machine. This is a simple but highly flexible approach.

2. Wallclock measurement of the time taken on the simulating machine can be used via a suitable multiplier (scale factor), to obtain the predicted running time on the target machine.

3. A better approximation is to use hardware performance counters on the simulating machine to count floating-point, integer and branch instructions (for example), and then to use a simple heuristic using the time for each of these operations on the target machine to give the predicted total computation time. Cache performance and the memory footprint effects can be approximated by percentage of memory accesses and cache hit/miss ratio.

4. A much more accurate way to estimate the time for every instruction is to use a hardware simulator that is cycle accurate model for the target machine.

The first three of the above described methods are currently supported in the simulator. The hardware simulator described in the last approach is being explored.

4.2 Predicting Network Performance

It is also necessary to simulate the network environment of the target machine to get the accurate performance prediction. The possible approaches are described below in the increasing order of accuracy (and complexity).

1. No contention modeling: the simplest approach ignores the network contention. The predicted receive time of any message will be just based on topology, designed network parameters and a per message overhead.

2. Back-patching: this approach stretches communication times based on the communication activity during each time period, using a network contention model.

3. Network simulation: this approach uses detailed modeling of the network, implemented as a parallel (or sequential) simulator.

The first approach is used in this paper. Although it may sound too simple, we found it models most computation bounded applications reasonably well. The network simulator is also being explored in [20].

5 Performance Simulation

In Section 3, we identified three kinds of synchronization overhead often found in simulators based on optimistic concurrency control. In performance simulation of an extreme-scale parallel machine, these overheads are prohibitive, in terms of both the CPU and memory costs. Our methodology in BigSim is to utilize the inherent determinacy in the application to reduce the synchronization overhead in simulation.

To introduce our parallel simulator concepts in a simple context, we first describe our simulator for a restricted class of programs. We then describe the generalization of this methodology for a broader class of the non-linear order parallel applications.

5.1 Simulating Linear Order Applications

A simple class of deterministic programs are those which permit messages to be processed in exactly one order. In the paper, we call them *linear order* parallel programs. For example, consider an MPI program that uses no wildcard receive and ensures only one message with a given tag and sender-processor is available at a time.

In linear order parallel applications, application messages are guaranteed to be delivered to the application in the expected order. The communication runtime handles any out-of-order message by buffering it until the application asks for it. The simulation is trivial in this case since no simulation event needs to be rolled back. Therefore, all overhead of checkpointing, rollback and re-execution can be avoided.

The synchronization algorithm for the simulation of such parallel applications is simple. As illustrated in Figure 2, a virtual processor timer($curT$) is maintained for each logical processor (LP) (implemented as a user-level thread in BigSim). When a message($m2$) of the program is sent out, we calculate its predicted receive time on the destination LP using a network model (Section 4). For example, in the simple model it is just the sum of the current thread

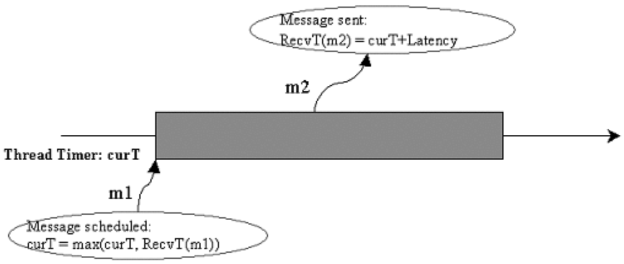


Figure 2. Timestamping events

time and the expected communication latency to the destination. The predicted receive time is then stored along with the message. When the *receive* statement for this message is executed($m1$) on the destination LP, its virtual processor timer is updated to the maximum of the current thread time and the predicted receive time of the message. Since there is only one order of execution possible, no rollback is necessary, hence no checkpoint is needed either.

5.2 Simulating a Broader Class of Applications

Linear order applications are limited in their expressiveness. For example, many parallel programs written in message-driven language such as CHARM++ do not belong to this category. In CHARM++, messages directed to an object can arrive in any order.

In the next subsection, we first describe a simple class of message-driven applications and their simulation, followed by a more complex and broader class of applications.

5.3 Simulation of Message Driven Programs

In message driven programs, the execution of a message is ready to be scheduled when the corresponding message invoking it arrives. In *atomic message-driven* programs, the execution is deterministic even when messages (method invocations) execute in different sequences on an object: either the object is providing information to other objects in request-response mode, or is processing multiple method invocations that complete as a set before the object (and the application) continues on to the next phase, possibly via a reduction.

Due to the deterministic property in method invocation, the simulation for such class of applications does not require checkpointing and re-executing an event, since re-execution of an event in the time of rollback will only produce the same states. In this case, rollback process is then simplified as *timestamp correction* of events in the simulation.

Figure 3 shows an initial timeline as an example. Each block represents an event in simulation, recording an execution of an application message. The header of each message stores its predicted arrival time (shown as *RecvTime* in Figure 3). A message can be executed on an LP if it is idle at the receive time. If the message arrives in the middle of an execution, it has to wait until the current execution finishes (see M5 in Figure 3). Assume that M4 has its predicted receive time now updated to an earlier time as shown in Figure 4(a). After timestamp correction, the modified timeline is shown in Figure 4(b). M4 is inserted back into the execution timeline with updated *RecvTime* and M5 is now executed right at its *RecvTime*. Note that all the events affected in the execution timeline (M4, M3, M5 and M6) also send out timestamp correction messages to inform all the spawned events about the changes in the timestamp.

Atomic message-drive programs are relatively rare, but they give us an opportunity to present the above timestamp correction approach in a simpler setting. A more complex class of applications, one that we call *non-linear* order programs, is a generalization of both atomic message-driven programs and linear order programs. Non-linear order programs have even more complex dependences, yet are essentially deterministic, as illustrated in the next subsection.

5.4 Non-linear Order Parallel Applications

Message-driven systems such as CHARM++ allow a process (or object, in case of CHARM++) to handle messages in multiple possible orders. This allows for a better overlap of communication and computation, since the system can handle the messages in the order they are received. However, from the point of view of simulation, this creates a complex causal dependence.

For example, take a 5-point stencil (Jacobi-like) program with 1-D decomposition, written in a message driven language: every chunk of data (implemented as parallel object) waits for a message from its left neighbor and the other from its right neighbor. As illustrated in the first timeline of Figure 5, the message from the right object invokes the function *getStripFromRight* on this object and the message from the left object invokes *getStripFromLeft*. These two messages generate events $e1$ and $e2$ in the simulator. When both messages arrive, the program calls the function *doWork* to do the calculation. Since the messages from left and right may arrive out of order, both functions need to be written in such a way that the later-invoked should call *doWork*.

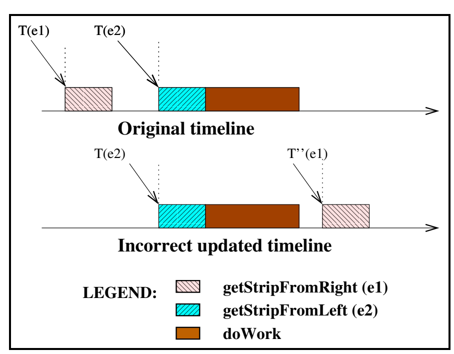


Figure 5. Incorrect correction scheme

When timestamp corrections are performed, the updated receive time of $e1$ may become $T'(e1) > T(e1)$. A naive application of the timestamp correction scheme described in the last section will move event $e1$ to a later point in time, as shown in the second timeline in Figure 5. Now the function *doWork* is called by the first arrived message, which is

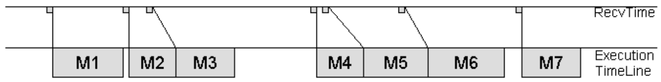
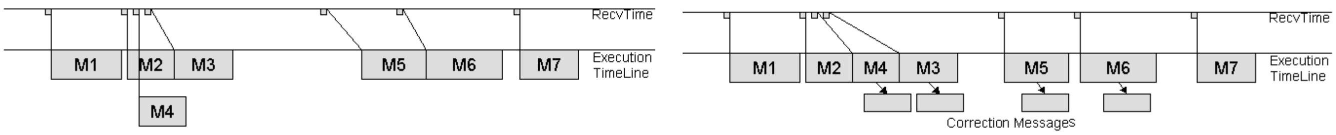


Figure 3. Initial timeline



(a) Receive time of M4 is updated to some earlier time

(b) Updated timeline

Figure 4. Timelines after updating event receive time and after complete correction

incorrect. This simple example shows that naive timestamp scheme can easily violate the dependencies among events.

Full checkpointing, rollback and re-execution of the event is one (very expensive) solution to ensure the correctness of the simulation. However, at a higher level, the program *is* deterministic — event corresponding to *doWork* depends on the completion of both events *getStripFromLeft* and *getStripFromRight* regardless of the order of their arrival. This determinacy of a program can be exploited to carry out the timestamp correction without re-executing the events (i.e. without having to re-execute the application code).

In fact, even from a programming perspective, such dependencies are problematic. Programmers have to write explicit code for maintaining counters, and flags (to see if both messages have arrived) and to buffer messages. Also, the flow of control is obscured by the split phases specification style. For these reasons, CHARM++ provides a notation called "Structured Dagger" that allows explicit representation of event dependencies. We propose to extract the dependency information from this language, along with runtime tracking of dependencies, to carry out deterministic simulation without re-execution.

5.4.1 Structured Dagger - a Language for Expressing Event Dependencies

Structured Dagger [9] is developed as a coordination language built on top of CHARM++. It allows a programmer to express the control flow within an object naturally using certain C language-like constructs.

In Structured Dagger, four categories of control structures are provided for expressing dependencies. They are *When-Block*, *Ordering Construct*, *Conditional and Looping Constructs* and *Atomic Construct*.

Figure 6 shows an example of the parallel 5-point stencil program with 1-D decomposition written in Structured Dagger. In the program, the **for** loop starts the iterations, which begin with calling *sendStripToLeftAndRight* in an **atomic** construct to send out messages to its neighbors [3]. The **overlap** immediately following asserts that the two events corresponding to *getStripFromLeft* and *getStripFromRight* can arrive and be processed in any order. The **when** construct simply says that when, for example, *getStripFromLeft* happens, it invokes the action in the **atomic** construct which calls a plain C++ function *copyStripFromLeft* to process the application message. When both events happen, function *doWork* in the last **atomic** construct will be invoked and the program enters the next iteration. Specifically, this sample code describes the event dependencies among events of application message *getStripFromLeft*, *getStripFromRight* and *doWork*.

[3]The atomic construct encapsulates any C language code

```
entry void jacobiLifeCycle()
{
  for (i=0; i<MAX_ITER; i++)
  {
    atomic {sendStripToLeftAndRight();}
    overlap
    {
      when getStripFromLeft(Msg *leftMsg)
         { atomic { copyStripFromLeft(leftMsg); } }
      when getStripFromRight(Msg *rightMsg)
         { atomic { copyStripFromRight(rightMsg); } }
    }
    atomic{ doWork(); /* Jacobi Relaxation */ }
  }
}
```

Figure 6. Sample code in Structured Dagger

The Structured Dagger program is compiled and translated into a normal C++ program with parallel runtime function calls. Interacting with the simulator using function API, the simulator gathers dependencies among events to ensure the correctness of the simulation. The new simulation scheme with Structured Dagger for non-linear order parallel applications is described next.

5.4.2 Simulating Non-linear Order Applications

With the help of language and runtime support, the simulator is able to capture the event dependencies which otherwise would be hidden in user programs. During the simulation, the simulator and the CHARM++ or AMPI runtime system work together to maintain the order of the executions according to the dependencies among events.

This approach also applies to a large class of MPI programs that use MPI_Irecv and MPI_Waitall as well: the *waitall* operation is simply recorded as having backward dependencies on all the pending *irecvs*. To make this happen, the runtime implementation of the MPI calls needs to interface to the simulator with the event dependency information. Such an interface is implemented in our AMPI.

In the simulations of both MPI and CHARM++ program, with the new scheme, the rollback process becomes greatly simplified. No event needs to be re-executed because re-execution of an event will only produce the same states. Thus, checkpointing of the states is also avoided since they are not needed any more. Rollback process in the new scheme is now simply an extension of the timestamp correction described in Section 5.1 under the constraints of the event dependencies. To illustrate the new simulation scheme better, we use the same jacobi example in Figure 6.

As the simulator runs, a chain of logs preserving the event dependencies is created on the fly. Every event E has a list of forward and backward dependents. The backward dependents of E will be those events which must complete before E can start. The forward dependents of E will be the list of those events that have E as one of their backward dependents. In the jacobi example, the event *doWork* has both *getStripFromLeft* and *getStripFromRight* as its backward dependents.

To preserve the order between the dependents, an event can only be allowed to execute after all the events that it depends on have been executed. To capture this we define a new term *effStartTime* (called effective start time) recursively as: $max(mEST, currentTime)$ where $mEST$ is the maximum *effStartTime* of all the backward dependents (zero if no backward dependents are present). The *effStartTime* is the time earlier than which the event can not start to ensure that we maintain the dependency relation between the events. The timeline will now be maintained in the non-decreasing order of the *effStartTime*.

6 Case Studies

We first present results of validation of BigSim on Lemieux [12] at Pittsburgh Supercomputing Center. We then present results of performance prediction and performance analysis of some real world applications for BlueGene/L. Finally, we will present the scaling performance of the BigSim itself.

6.1 Validation

In order to validate our BigSim, we compared the actual running time of a 7-point stencil program with 3-D decomposition (Jacobi3D) written in MPI with our simulation of it using BigSim. In the program, every chunk of data communicates with its six neighbors in three dimensions. After Jacobi relaxation computation, the maximum error is calculated via **MPI_Allreduce** of all local errors.

The result is shown in Table 1 for a problem with a fixed size in all the runs. The first row in the table shows the running time on 64 to 512 processors; the second row shows the predicted running time when simulating these processors using only 32 real processors. It shows that our simulated execution time generally agrees with the actual execution time to within about 6% although a simple latency based network model is used.

Processors	64	128	256	512
Actual run time (s)	1.072	0.481	0.259	0.145
predicted time (s)	1.046	0.512	0.270	0.155

Table 1. Actual vs. predicted time

6.2 Jacobi on BG/L

With BigSim, we are now able to study the performance issues of some real world applications on a machine before it is built, specifically BlueGene/L in the following case studies.

To facilitate performance analysis for applications on this machine, *Projections*, a postmortem performance analysis tool associated with CHARM++ has been ported to BigSim. It provides the capabilities of detailed event tracing and interactive graphical analysis.

The Jacobi program written in CHARM++ and Structured Dagger (Sec 5.4.1) was used as a case-study to further analyze and verify the simulator. In this simulation, the network model uses a per-hop and per-corner latency of 5ns and 75ns respectively. For experiments with different network configurations, the network latency can be increased by scaling both the per-hop and per-corner latency by the same factor.

The timelines generated by Projections are shown in Figures 7 and 8 for a selected subset of 64,000 simulated processors. Figure 7 was generated without simulation (the program was only emulated), while Figure 8 was generated with simulation. The separation between the events in Figure 7 is caused by the direct or cascaded effect of the out-of-order delivery of messages. As we can see in Figure 8, the timestamps of out of order messages were corrected and the gaps disappeared.

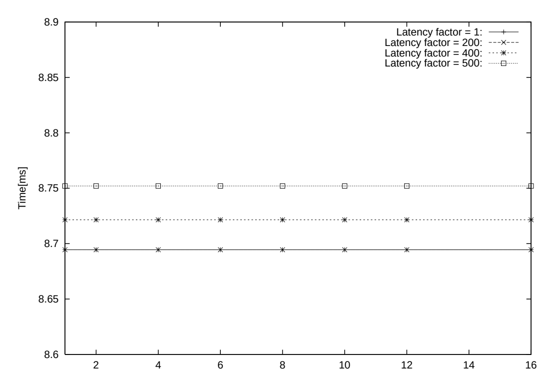


Figure 9. Predicted time vs real processors

We also validated the simulator by comparing the result with expected behaviors. The results that we obtained are summarized as follows:

- For a valid timestamp correction scheme we expect same predicted time for the same problem independent of the number of real processors used for simulation. This can be used to verify the simulator. Predicted performance was indeed found to be same across different

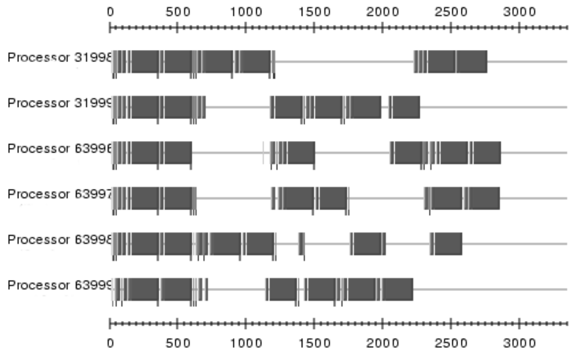


Figure 7. Timelines before correction

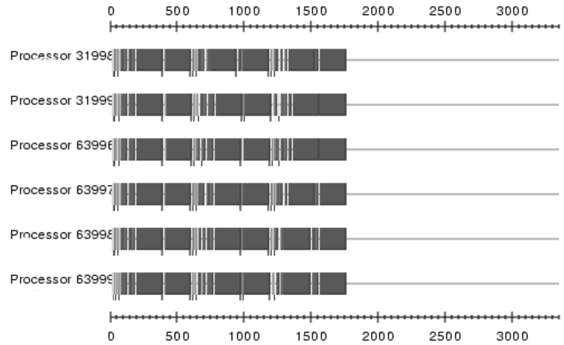


Figure 8. Timelines after correction

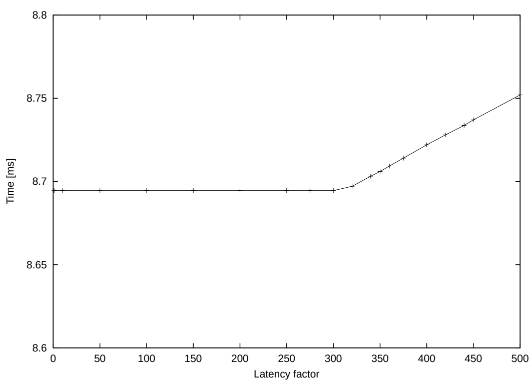


Figure 10. Predicted time vs latency factor

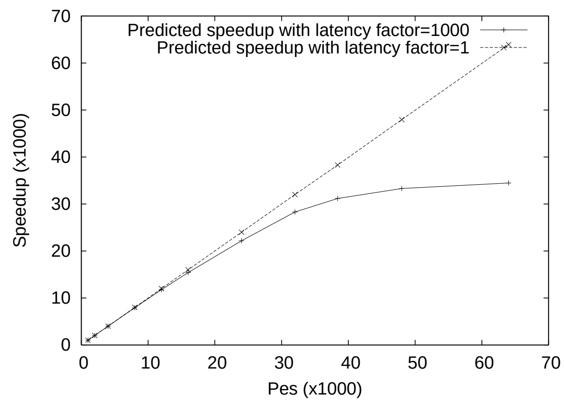


Figure 11. Speedup for Jacobi1D

runs and the result for Jacobi is shown in Figure 9 with different network latency configurations.

- As we increase the network latency we expect the predicted time to remain constant up to a limit and increase thereafter, due to overlap of computation and communication. Note that the predicted time was measured as a function of multiplicative factor by which the per-hop and per-corner latencies are increased. The result was as expected and is shown in Figure 10.
- The speedup was also measured based on the predicted time for different latency factors as shown in Figure 11. For a very low network latency, the speedup was found to be close to linear, and dropped as the latency factor was raised. This is because when the number of simulated processors increases, the work per-processor reduces as the computation can not makeup for communication delay, reducing the speedup.

6.3 Molecular Dynamics Simulation

The molecular dynamics simulation of biomolecules is one of the planned applications for BlueGene/L. It is a challenging application to parallelize. A microsecond simulation includes about a billion timesteps. Thus, each timestep involves a relatively small amount of computation that must be effectively parallelized. We have developed **NAMD**, a Gordon Bell award winning parallel molecular dynamics application that is also written in CHARM++. Although it has been shown to scale to 3000 processors [11], it is not ready for extreme-scale parallel machines due to the relatively limited parallelism exploited in the application.

In NAMD, the atoms in the simulation are divided spatially into cells roughly the size of the cutoff distance. Local interactions are calculated for each timestep between only the nearest neighbor cells ("one-away" interactions), as illustrated in Figure 12. This ensures that all atoms within the cutoff radius are calculated. However, this strategy produces a division that is coarsely grained for planned machines such as BlueGene/L. For example, with a cutoff radius of 15 Å, a 150 x 150 x 150 Å simulation space would give only 1,000 cells and 13,000 cell-to-cell interactions [4] to calculate. Considering that the BlueGene/L machine is approximately 64,000 nodes, the division would leave nodes idle even if interactions were delegated to a single node.

To address the issue of creating finer-grained parallelism for cutoff interactions, **LeanMD** is being developed as an experimental code. In LeanMD, the "one-away" strategy is replaced with a "k-away" strategy. Instead of one cell representing the cutoff distance, in LeanMD three cells would

[4] 1,000*27/2, since cell-to-cell forces are symmetric.

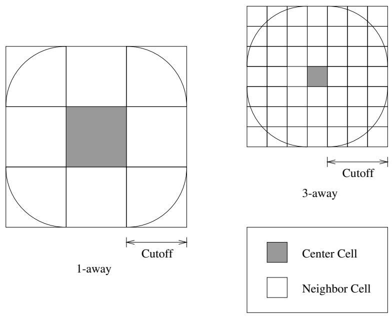


Figure 12. 1-away and 3-away cut-off distance

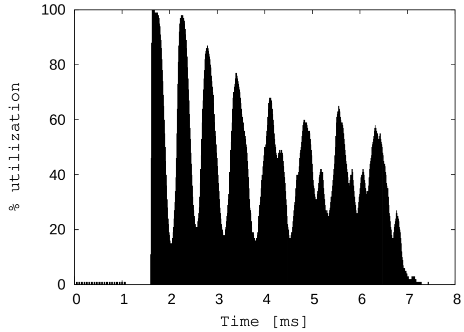


Figure 13. Average utilization per interval for Molecular Dynamics on 32,000 processors

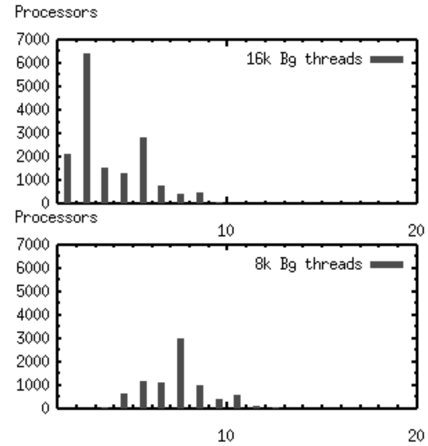


Figure 14. Distribution of processors based on load in ms

span the cutoff distance as shown in Figure 12. Therefore, in order to do the cutoff calculation, a cell must compute its interactions with every cell that is "three-away" in this scenario. Given the simulation example above, a three-away strategy would produce 27,000 cells and more than 4 million cell-to-cell interactions, a number of objects that can be easily distributed across the 64,000 nodes of BlueGene/L.

We have been able to run LeanMD (a full-fledged CHARM++ code) on our simulator on Lemieux as a real benchmark. We have run 3 away ER-GRE benchmark which consists of 36573 atoms, with a cutoff of 12 Å, the cell size thus is 4x4x4 and the simulation space is 23x23x23 cells. The number of cell-to-cell interactions is more than 1.6 million. We simulate the BlueGene/L nodes of size from 1K to 64K of full machine size. The predicted speedup is shown in Figure 15 by the bottom curve.

The simulation data can be used to carry out more detailed performance analysis. The average processor utilization, as it varies with time, is shown in Figure 13 for 32k simulated processors. The utilization stabilizes at about 50%, but rises and falls within each timestep. This corresponds to the speedup saturation seen in Figure 15. This could be due to either communication latencies, critical paths or load imbalance. To understand the saturation of the speedup we used the performance logs to calculate the load on individual processors. Figure 14 shows a histogram of this data in the case of 8k and 16k simulated processors. Although about 6000 out of 16000 processors have a load of about 2ms, a few are seen to have a load as high as 8ms. This suggests that load balance is a major performance issue. To understand what portion of performance loss is explained by load imbalance alone, we plot the estimated speedup($P * avgLoad/maxLoad$) based on load imbalance loss alone (top curve in Figure 15) and compare it with simulated speedup. The closeness of both curves confirms that load imbalance is the primary cause of performance loss. Only at 64K processors do the curves deviate, indicating influence of other factors such as communication

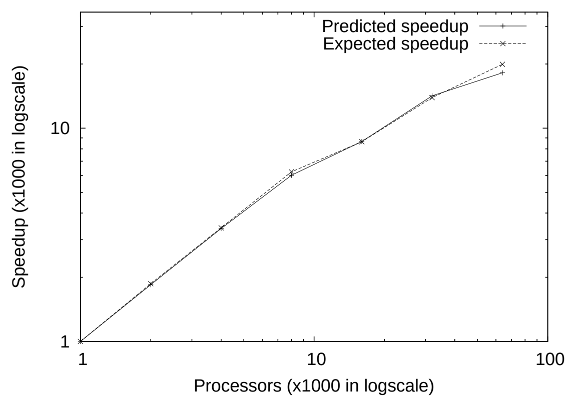


Figure 15. LeanMD predicted and expected speedup for up to 64,000 processors

overhead or critical paths. Such detailed performance analysis is possible because of the rich performance trace data produced by the simulator. [5]

For these simulations we used a no-contention communication model, with possibly too optimistic communication parameters. We plan to implement a more detailed network model [20] and get realistic network parameters from IBM for the case of BlueGene/L. Preliminary case-studies demonstrate that the simulator can be used to identify performance issues for scaling individual applications.

6.4 Performance of the Simulation

We also measured the performance of the simulator itself using LeanMD as a sample application. We demonstrate the scalability of the parallel simulator in Figure 16. The simulation was found to scale reasonably over hundreds of processors. The efficiency of the simulation depends on the

[5] This also affects to the scalability of **Projections**. It is routinely used for analysis of a few thousand processors on real runs, but we are able to use it, unmodified, for 64K processors.

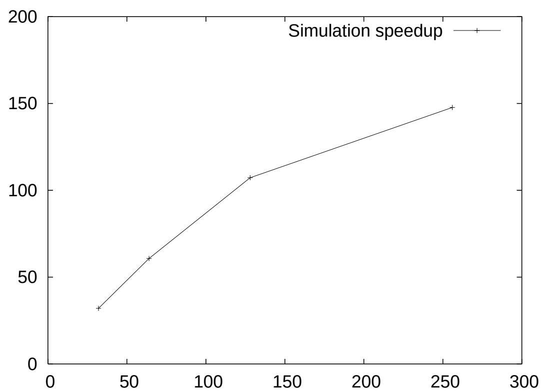


Figure 16. Simulation Speedup

number of correction messages sent. In one simulation, correction and real messages sent were compared for different simulated processors as shown in Table 2. The low ratio of correction messages to real messages was encouraging. This typically leads to only about 50% overhead for simulation compared with emulation alone.

Processors	8k	16K	32k	64k
Real Msgs	20.04M	20.18M	20.42M	20.93M
Corr. Msgs	357351	305487	126629	59762

Table 2. Proportion of correction messages

7 Conclusion and Future Work

Although parallel simulation for extremely large parallel machines (at least tens of thousands of processors) is very challenging, we have shown that by utilizing the inherent determinacy of parallel applications and tightly coupling the simulator with emulator at runtime, we are able to improve the simulation efficiency by reducing the synchronization overhead often found in PDES. The BigSim parallel simulator we have developed is capable of making performance predictions for a broad class of applications on very large parallel machines. The online mode of the simulator is also useful in studying various performance issues in parallel applications, such as load balance and fault tolerance issues. The explored simulation techniques show good parallel scalability. Future work will focus on increasing accuracy by incorporating an instruction level simulator and a network simulator.

References

[1] à la carte - a Los Alamos Computer Architecture Toolkit for Extreme-Scale Architecture Simulation. http://wwwc3.lanl.gov/ parsim.

[2] ASCI Purple RPF Home Page. http://www.llnl.gov/asci/purple.

[3] An Overview of the BlueGene/L Supercomputer. In *Supercomputing 2002 Technical Papers*, Baltimore, Maryland, 2002. The BlueGene/L Team, IBM and Lawrence Livermore National Laboratory.

[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, 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.

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

[7] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto. Time warp operating system. In *Proceedings of the 11th ACM Symposium on Operating System Principles*, pages 77–93, 1987.

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

[9] L. V. Kale and M. 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.

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

[11] L. V. Kalé, S. Kumar, G. Zheng, and C. W. Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop (ICCS'03)*, Melbourne, Australia, June 2003.

[12] Lemieux. http://www.psc.edu/machines/tcs/lemieux.html.

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

[14] 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, September 2002.

[15] M. Potnuru. Automatic out-of-core exceution support for charm++. Master's thesis, University of Illinois at Urbana-Champaign, 2003.

[16] N. Saboo, A. K. Singla, J. M. Unger, and L. V. Kalé. Emulating petaflops machines and blue gene. In *Workshop on Massively Parallel Processing (IPDPS'01)*, San Francisco, CA, April 2001.

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

[18] J. S. Steinman. Breathing time warp. In *Proceedings of the 7th workshop on Parallel and Distributed Simulation*, pages 109–118. ACM Press, 1993.

[19] G. Zheng, A. K. Singla, J. M. Unger, and L. 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.

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