© Copyright by Gunavardhan Kakulapati, 2003

SIMULATING LARGE PARALLEL MACHINES FOR PERFORMANCE PREDICTION

BY

GUNAVARDHAN KAKULAPATI

B.Tech., Indian Institute of Technology, Madras, 2001

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2003

Urbana, Illinois

Abstract

The masters thesis describes a simulator for large parallel machines like BlueGene/L that provides the ability to make performance predictions, based on actual execution of real applications.

It is useful to evaluate the performance of many applications on such machines even before they are built. In this thesis, we present an online parallel simulator which is based on a parallel discrete event simulation scheme and also demonstrate sequential and parallel post-mortem simulation schemes. The techniques for optimizing online parallel simulations of machines with large number of processors on the ones with fewer number of processors are also explored. Finally, a comparison of online and post-mortem approaches is also presented.

To my parents.

Acknowledgements

First of all I would like to thank my advisor, Prof. Laxmikant Kalé for his encouragement, support and motivation. I also thank him for the knowledge and expertise that I gained by working under his guidance. The projects assigned to me perfectly matched my interests and it was a lot of fun working with him and being in his group.

I thank Gengbin Zheng for the enormous help that he has provided me on this project. Working with him was a very rewarding experience. This project would not have been possible without his inputs. I also thank Orion Lawlor for his help on many projects that I worked on. A very special thanks to all the members of the Parallel Programming Laboratory especially Cheewai and Chao for all the fun times I had in the lab.

Finally, I would like to thank my parents and my brother for all the support and encouragement that they have always given me.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Many large parallel machines are now being designed. The BlueGene (BG/L) machine being built by IBM will have 65536 dual-processor nodes with a peak performance of 360 teraFLOPS and is scheduled to be operational in the 2004-2005 timeframe. Other large parallel machines will be also expected to be ready in near future. ASCI Purple will have 12k processors, ASCI Red Storm will have 10k-30k processors. A design by IBM called the Cyclops may have over one million floating point units, fed by 8 million instruction streams supported by individual thread units. In response to an initiative by the U.S. Department of Defense, newer architectures are coming up for machines to be built within the next few years [1].

It is essential to evaluate the performance of such machines for different target applications before they are built or before they are actually purchased. It is not sufficient to multiply the peak floating point performance by the number of processors. Communication performance, application characteristics and the behavior of run-time support systems contribute to the overall performance.

Even for existing large machines, our performance prediction approach is useful. Time on large machines is hard to get and must be reserved well ahead. However, every time

[1] From news article at: http://www.hoise.com/primeur/01/articles/monthly/AE-PR-07-01-13.html

a performance measurement run is carried out, application developers must spend time visualizing and analyzing the performance data before the next set of optimizations are decided upon. Waiting time for the next running slot is atleast a day. With a simulator, this performance debugging cycle can be shortened considerably.

1.2 Thesis objectives

The new machines can be evaluated by running planned applications on full-fledged simulators of such machines. The approach presented here is a step in the direction of accurate simulations of extremely large parallel machines using currently available parallel machines.

The simulator builds on an already existing system that can emulate large parallel machines [7]. Based on the Charm++ [2] parallel programming system, the emulator has successfully simulated 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 and does not provide performance information.

To make performance predictions, we explore three alternative (and competing techniques):

- Modify the emulator to carry out an online Parallel Discrete Event Simulation (PDES).
- Record traces during emulation and run a sequential trace-driven simulation.
- Record traces during emulation and run a parallel simulation based on the generated traces

The thesis explores the above mentioned strategies and also identifies the advantages of each of these schemes. In general, large traces and slower simulations make the sequential approach infeasible. The other approach is a parallel post-mortem scheme also for which the traces have to be recorded. The online parallel simulator lets the execution of the

program proceed as usual. It concurrently runs a parallel algorithm to correct timestamps of individual messages. We present the parallel approach in the context of a simple, evidently deterministic, class of applications called linear-order programs and then generalize it for a broader class of applications.

1.3 Thesis organization

We first give a description of the emulator and then the overview of the PDES approach that is taken. Later in section 4 we describe the Structured Dagger coordination language that can be used to capture event dependencies. Section 5 describes the online correction scheme. We then present optimization techniques that speed up the parallel simulation, along with some performance results. A few simple application benchmarks that illustrate the use of the simulator, predicting performance on a BG/L like machine with 64K processors are demonstrated. Section 7 explores both the parallel and sequential post-mortem methods. Section 8 concludes by describing the directions of further research that emerge from the thesis.

Chapter 2

Description of the Simulator

Before exploring the various simulation strategies which is the main focus of this thesis, we describe the components of the simulator.

In order to simulate a future machine with extremely large size on an existing parallel machine with hundreds of processors, one physical processor has to simulate hundreds or even thousands of processors of a simulated machine. Each simulated processor is represented as a thread on a physical processor. The total memory requirement for the simulation is an important consideration. Fortunately, some of the planned PetaFLOPs class machines have low memory-to-processor ratio. For example, Blue Gene/C was originally designed to have about half a terabyte of total memory. Thus, to emulate Blue Gene/C running an application which uses the full machine will require 500 processors of a traditional parallel machine with 1G memory per processor. Also, many real world applications such as molecular dynamics simulation do not require large amount of memory. For applications that do require large amount of memory, it is still possible to use automatic out-of-core [6] execution to temporarily move the data in memory to disk when it is not needed immediately. This swapping increases the simulation time but the predicted running time on the target machine is unaffected. The Parallel Simulator builds on an emulator for PetaFLOPs class machines [1]. We first describe the emulator, and the component models for estimating costs of computation and communication.

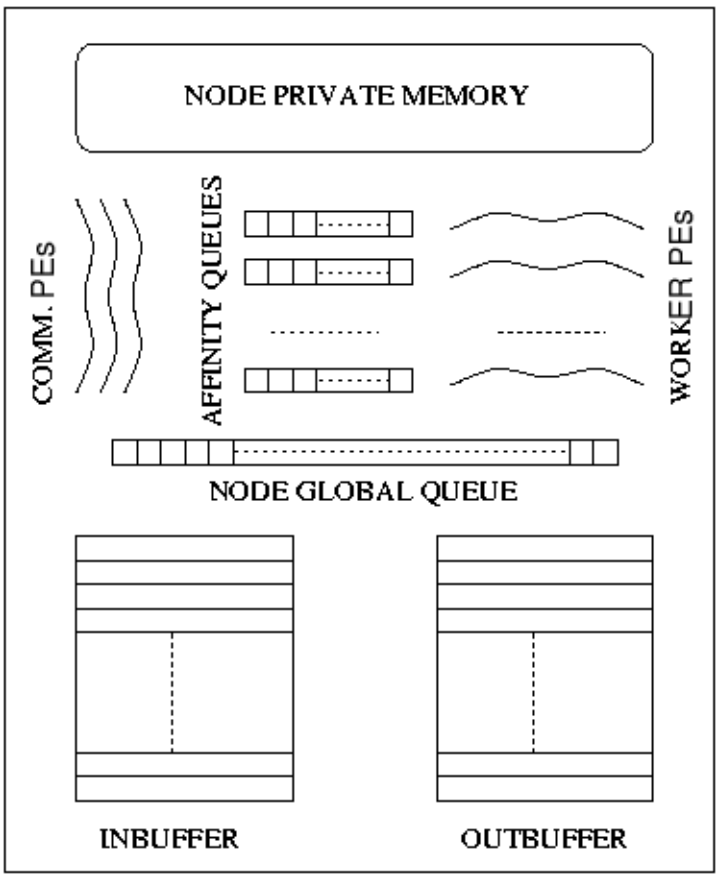

Figure 2.1: Functional view of an emulated node

2.1 Emulator for parallel machines

The emulator [7] supports a low-level API. The emulator is implemented in Converse [4]. Though the API mimics the Bluegene API, it is fairly general to support other architectures. API supports multiple instruction streams (hardware threads) that share memory in each single node. An emulated node as shown in 2.1 has worker threads and communication processors. The node has a global queue and an affinity queue for each worker thread. The node design is general enough to encompass other architectures that may have different number of processors and co-processors in each node. The emulator however cannot make any performance predictions or give an estimate for the running time on the real machine.

2.2 Component performance models

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

2.2.1 Predicting the time of sequential code

The walltime taken to run a section of code on traditional machines cannot be directly used to estimate the compute time on the target machine. As we don't 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).

- 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.
- 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.
- 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.
- 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 simulator currently supports the first three of the above described methods.

2.2.2 Predicting performance messaging

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).

- 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.

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

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

The simulator currently supports only the first two approaches.

Chapter 3

PDES for Simulating Linear Order Applications

In general, simulation for performance predictions can be carried out as a Parallel Discrete Event Simulation, or PDES. All entities have a state which changes over time. The changes of state are known as events. The discrete point in time at which the event occurs is a timestamp. A discrete event simulation models these state changes. Consider a simulation of an application using a message passing or message driven system The entities include emulated processors and all components in the emulated 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.

3.1 Timestamping of messages

A virtual processor time ($curT$) is maintained for each simulated processor (implemented as a user level thread in the simulator). The message delivery is simulated using timestamped events. Each message carries a predicted receive time that denotes when the message would be delivered at its destination. The predicted time is the sum of the current thread time and the expected communication latency. When the message receive statement for this message

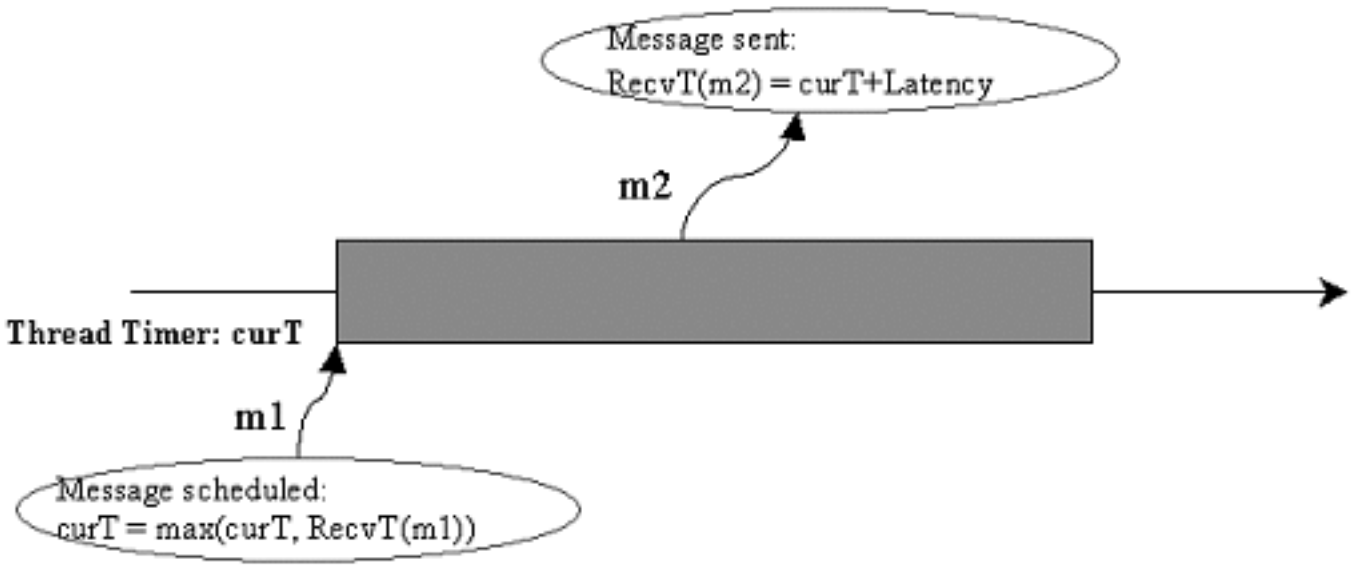

Figure 3.1: Timestamping different events

is executed, the thread timer is updated to the maximum of the current thread timer and the predicted receive time of the message. Figure 3.1 shows this timestamping of messages. Every event has a *recvTime* which is the time when the message that triggers this event is received. The start time of the event will always be greater than the *recvTime*.

3.2 Sequencing constraints

Discrete Event Simulations are difficult to parallelize efficiently because of the issues of causality and sequencing. In the parallel simulation, each LP works on its own by selecting the earliest event available to it and processing it. Doing this can result in sequencing errors caused by out-of-order message delivery. Out-of-order message delivery occurs when the messages are delivered in an order different from the one expected on that machine. If a message $M1$ with an earlier timestamp T(M1) arrives later than another message $M2$ with timestamp T(M2) with T(M1) $<$ T(M2), then $M1$ will be executed in the context of the future which is wrong.

Causality errors also have be handled. Suppose event E1 triggered by message M1 on

LP1 and and event E2 by message M2 on LP2 proceed concurrently where T(M1) < T (M2). And E1 sends a message M3 to LP2 triggering an event E3, with T(M3) < T(M2). Then M3 must be executed ahead of M2 to ensure the correct order of events. Timestamp correction is needed to handle such errors, the details of which are described later in the thesis.

3.3 Possible strategies

The possible strategies for execution of events without violating sequencing can be broadly classified as conservative and optimistic. For the conservative approach, one has to ensure the safety of an event globally before processing it. Determining this safety is expensive and can reduce potential parallelism. The optimistic approach allows the execution to go ahead and correct the sequencing violations. This is performed by doing rollbacks and re-executing the old events after correcting the order of arrival of messages. This approach can be expensive when directly applied. We later present an optimized optimistic scheme that exploits the inherent determinacy of the parallel programs avoiding rollbacks and checkpointing.

3.4 Classification of applications

Some parallel programs give a deterministic result even in the presence of the out-of-order messages. Executing such a program any number of times will give the same result even though messages maybe processed in different across those executions. With a few exceptions (such as branch-and-bound and certain classes of truly asynchronous algorithms), parallel programs are written to be deterministic. The results are same even though the execution orders of some components may be allowed to differ as they carry out the same computations. Non-deterministic programs are beyond the scope of this simulator as changing the order of messages will change the result itself and will require a total re-execution of the program. For the deterministic case, how the dependencies between events are captured and out-of-order

messages are corrected is described in the later chapters. Linear order applications are those that have all the messages processed in the same order in all of their executions and are simplest to simulate of the deterministic applications. This strict determinacy condition is easily exploited in the simulator. The class of non-linear order applications are those where this condition is not necessarily true. This class subsumes the linear-order applications.

3.5 Simulating linear order applications

In linear-order applications the messages are guaranteed to be used in a fixed order by the application. Across the multiple runs of the program that messages will be processed in the same order. The communication runtime handles any out-of-order messages by buffering them until the application asks for them. And the application always asks for the messages in the same order. This case is the simplest because messages will always be executed in their timestamp order. Simulating would only involve updating the current time based on the timestamp of the message that is about to be processed. This class of applications guarantees determinacy as it permits the messages to be processed in exactly one order. Examples of these are MPI programs that dont use irecv-waitall pairs. This is taken advantage of in performance simulation as no correction of timestamps or rollbacks are needed.

Chapter 4

Simulating Broader Classes of Applications Using Timestamp Correction

Although linear order programs guarantee determinacy and make it possible to simulate them without application-level rollbacks, they are limited in their expressiveness. For the more general class of non-linear order parallel applications, the different orders in which messages are received lead to different orders of execution.

4.1 Message driven programs and Charm++

Charm++ is an object-based portable parallel programming language that embodies message-driven execution. Charm++ consists of parallel objects and object arrays. These objects communicate via asynchronous method invocation. In Charm++, an entry method of one parallel object is executed when there is a method invocation (message) directed to it. In message driven programs, the execution of an event is ready to be scheduled when the corresponding message that invokes it arrives. In most message-driven programs, the execution is deterministic even when messages (method invocations) execute in different sequences on an object. Due to the deterministic property in method invocation, the simulation for such type of applications does not require rollback and checkpointing. However, timestamp correction

is needed to handle out of order message delivery.

4.2 Timestamp correction

The $recvTime$ of a method-invocation event is defined as the time when the message invoking that entry method is received on the destination processor. The terms $startTime$, $endTime$, and $execTime$ describe the times when the method execution starts, ends and how long it runs.

When an event is executed, its $startTime$ is computed by: $max(recvTime, currTime)$, where $currTime$ is the time on the executing processor. After an event is added to the timeline the $currTime$ is updated to the $endTime$ of that event. An event in the waiting queue can be scheduled as soon as the current event in the timeline is done and its invoking message has arrived. If the message to invoke an event E_a arrives during the execution of another event E_b, it has to be executed after E_a finishes, thus $startTime(E_a) = endTime(E_b)$. If the message arrives some time after E_b finishes, the $currTime$ is updated to the $recvTime$ of the message which is also the $startTime(E_a)$.

In our timestamp correction scheme, the $recvTime$ of a method-invocation event is defined as the time when the message invoking that entry method is received on the destination processor. The terms $startTime$, $endTime$, and $execTime$ describe the times when the method execution starts, ends and how long it runs. In Charm++, the events are non-preemptive, so the following always holds true: $execTime = endTime - startTime$. When an event is executed, its $startTime$ is computed by: $max(recvTime, currTime)$, where $currTime$ is the current thread time. After an event is added to the timeline the $currTime$ is updated to the $endTime$ of that event. An event in the waiting queue can be scheduled as soon as the current event in the timeline is done and its invoking message has arrived. If the message to invoke an event E_a arrives during the execution of another event E_b, it has to be executed after E_a finishes, thus $startTime(E_a) = endTime(E_b)$. If the message arrives

some time after E_b finishes, the $currTime$ is updated to the $recvTime$ of the message which is also the $startTime(E_a)$.

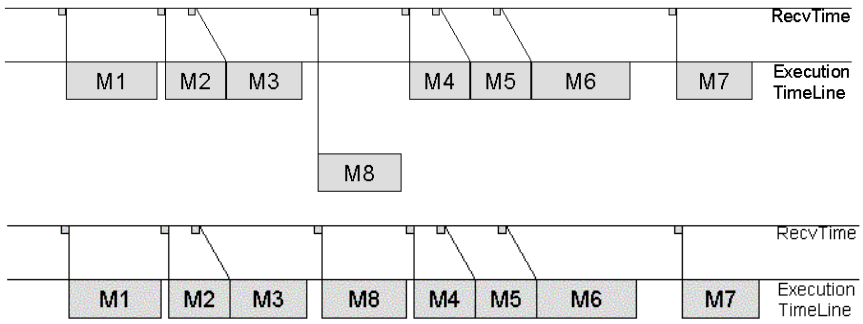

Figure 4.1: A simple case in timestamp correction

Timestamp correction scheme is needed to correct out-of-order delivery of messages. The first correction will be triggered if a message M_1 with an earlier timestamp arrives later than message M_2. In that case M_1 will be executed later (that is in the context of the future) even if its `recvTime` is earlier than M_2. This will be corrected by reordering the events whenever an insertion of an event breaks the `recvTime` order. Whenever an event `startTime` changes due to such a rearrangement, it sends out a correction message informing of this change, for every message that it sent during its execution. The proposed scheme reorders the events in the timeline as their `recvTime` changes. Several cases can arise as the events are moved along the timeline due to their updated receive times. In this approach the timeline will always have all its events in the non-decreasing order of their receive times.

The simplest example is shown in Figure 4.1. The event M8 arrives after M7, but has a timestamp before M4. Since the processor was idle between the processing of M3 and M4, and the idle time was adequate to accommodate the simulated execution time of M8, it will have its recvTime updated and will be inserted into the timeline accordingly. In this case the other events are not affected. If there are any events that were triggered by the messages sent by M8, they should be made aware of this change and should be updated. This is done using the correction messages.

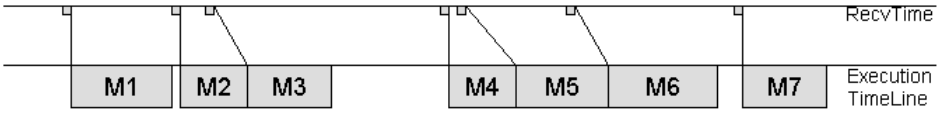

Figure 4.2: Initial timeline

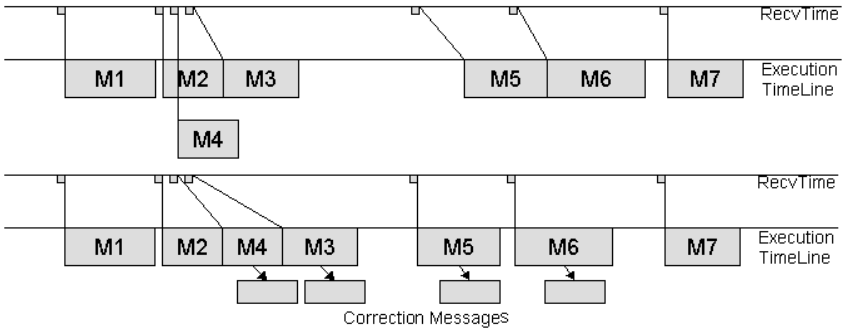

Figure 4.3: Case I (a) (b): Timelines after updating event receive time and after complete correction

Figure 4.2 shows an initial timeline. Figure 4.3(a) shows that event M4 has its receive time updated to a smaller value. The modified timeline is shown in figure 4.3(b). Note that the events that have their start time changed due to this shift send out correction messages to update other events that have been triggered by them. The figures 4.4(a),(b) demonstrate the case where the new receive time of event M4 is greater than its previous receive time. Note that in this case, after M4 is corrected, not only the events after M4 but the ones before M4 (M5,M6 in this case), send the correction messages. This is because if M4 were not present ahead of events M5 and M6, they would have been scheduled earlier to process at their `recvTime` as shown in figure 4.4(b). These corrections messages may cause some other events to be rearranged.

But this scheme cannot be directly applied to all non-linear parallel applications where the execution of a message is not *atomic*, for example, execution of an event depends on the occurrence of another event. Inherent dependencies between the events limit the current scheme as we demonstrate in the next section.

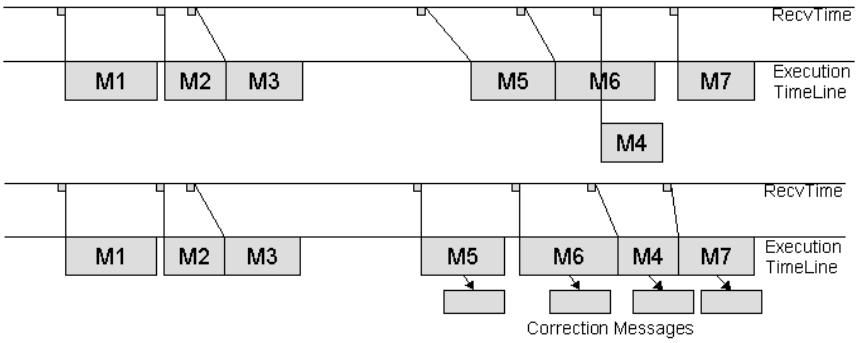

Figure 4.4: Case II (a) (b): Timelines after updating event receive time and after complete correction

4.3 Naive timing correction

Unexpected messages that arrive in Charm++ are buffered and executed when an entry function associated with it becomes eligible for execution. Suppose event *e3* is generated only after events *e1* and *e2* are executed. By a naive correction scheme if the timestamp of the message that triggers *e1* is changed to *T'(e1)* such that *T'(e1) > T(e1)* then the event *e1* is also moved to a later point of time without updating *e3* which is incorrect. This is demonstrated in the Fig 4.5.

The fundamental problem here is that the dependency among events is handled implicitly in user application. The dependency is determined by some states that are kept inside the user application. The key to the solution is to let the application expose the dependency of the events to the communication runtime instead of handling or hiding the dependency in the user application itself. This allows the runtime to extract the event dependency information from the application so that it knows how to deliver the events in the order that is expected.

4.4 Structured Dagger

Structured Dagger is developed as a coordination language built on top of Charm++. It allows a programmer to express the control flow within an object naturally via certain con-

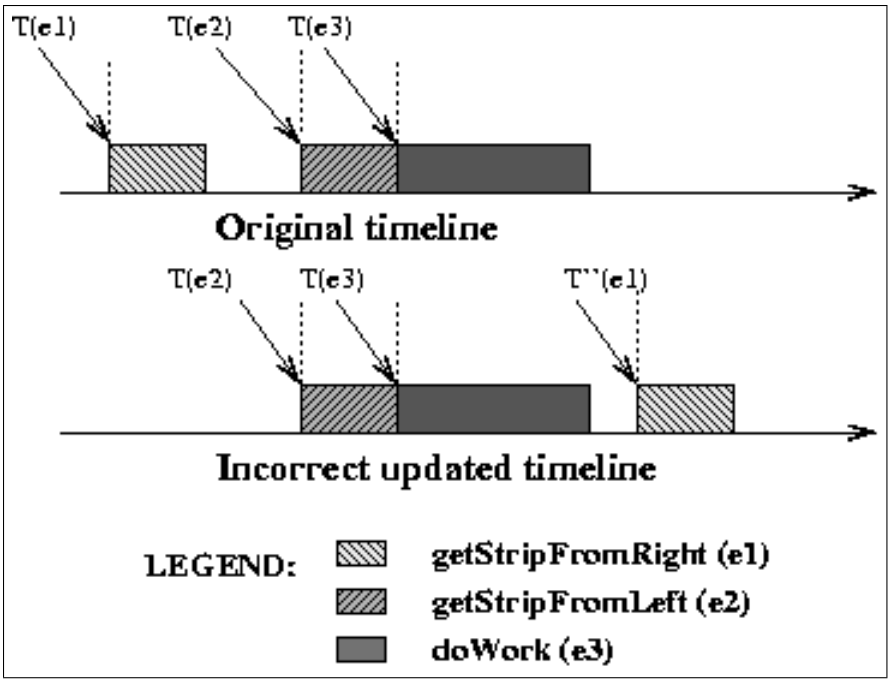

Figure 4.5: Incorrect correction scheme

structs, while overcoming limitations of thread-based languages without losing the performance benefits of adaptive message-driven execution. In other words, Structured-Dagger is a structured notation for specifying intra-process control dependences in message-driven programs. It combines the efficiency of message-driven execution with the explicitness of control specification. Structured Dagger allows easy expression of dependences among messages and computations and also among computations within the same object using when-blocks and various structured constructs. The Structured Dagger is adequate for expressing control-dependencies that form a series-parallel control-flow graph [1].

In Structured Dagger, constructs like When-Blocks, Ordering Construct, Conditional and Looping Constructs and Atomic Construct are provided. Use of these intuitive constructs are illustrated in the Figure 4.6.

As an example, we have shown a simple Jacobi program with 1D decomposition. The parallel Jacobi1D program uses a one-dimensional blocked distribution of data which is

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

Figure 4.6: Sample structured dagger code

divided into chunks. Each Chunk can be implemented as a Charm++ parallel object. The life cycle of this object `jacobiLifeCycle` can be written in structured-dagger program as shown in Fig 4.6. Entry function jacobiLifeCycle() is defined as special remotely invocable function of the Chunk object. Chares concurrent objects in Charm++ whose methods can be remotely invoked. When the messages from chares possessing the neighboring strips on the left and right arrive, the methods `getStripFromLeft` and `getStripFromRight` are triggered respectively. Only when both these messages have arrived, the computation can be performed by the `doWork` function. For the example in Figure 4.5, the dependency among events $e1$, $e2$ and $e3$ can be expressed explicitly as: `when` $e1$ and $e2$ { $e3$ }.

4.5 Event dependencies

The Structured-Dagger as part of Charm++ programming system has been ported to our emulator. The performance prediction study exploits Structured-Dagger's ability for expressing control-dependencies in application. With Structured Dagger, the runtime can capture the dependencies between events even when the object allows them to be processed in mul-

tiple orders. 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.

We use Structured Dagger coordination language to build the event dependencies, which will be accounted for when the events are reordered. In the example of Jacobi1D, relation between `getStripFromLeft`, `getStripFromRight` and `doWork` events can be captured using the overlap and when constructs as shown in Figure 4.6. As the structured dagger code runs, a chain of logs preserving the event dependencies is created on the fly. In the new strategy, every event E has a list of forward and backward dependents. The backward dependents 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 previous example, the event `doWork` has both `getStripFromLeft` and `getStripFromRight` as its backward dependents. In order to preserve the order between the dependents, an event can be added to the timeline only after all the events that it depends on have been added. To capture this we define a new term `effRecvTime` (called effective receive time) recursively as: $max(mERT, recvTime)$, where the term $mERT$ is the maximum `effRecvTime` of all the backward dependents (zero if no backward dependents are present). The `effRecvTime` is the time earlier to which the event cannot start to ensure that we maintain the dependency relation between the events. The startTime of an event will now be computed as $max($`effRecvTime`$, currTime)$. A correct timeline should be sorted based on the `effRecvTime`, instead of the recvTime. Simulations that use these dependencies subsume linear-order applications because those applications do not cause any causality violations or need any timestamp correction.

Chapter 5

Online Parallel Simulation: Approach and Optimizations

The online parallel timestamp correction scheme and results presented in the next chapter were developed in collaboration with Gengbin Zheng [9] and are also a part of his ongoing PhD. thesis research. The online parallel timestamp correction scheme is closer to the optimistic approach of Parallel Discrete Event Simulation. The application is run and whenever a late message with an earlier timestamp arrives, it has to be inserted in its correct place in the timeline. This causes more corrections by the events which have already been executed.

5.1 Description of the scheme

The idea of the timestamp correction was described earlier in Chapter 4. The online simulator uses event dependencies to overcome the limitations of the naive correction scheme. As the program proceeds and timestamps corrections arrive, the effRecvTime of many events change. This may cause reordering of the timeline. These events are rearranged in the order of their effRecvTime. The following steps describe this scheme:

(1) Calculate the earliest affected event (EAE) in the timeline

(2) Remove all events from the earliest affected event into R

(3) Initialize effRecvTime of those events to infinity

(4) Recalculate effRecvTime for all events in R whose backwardDeps are in not R

(5) While R is not empty

(6) Choose the least effRecvTime event from R

(7) Reinsert into timeline

(8) Update the effRecvTime of forward dependents of that event

When an event gets a new effRecvTime, the EAE in (1) is the earlier of its new position in the timeline and the current position. The above sequence of operations is performed whenever a correction message arrives. After processing it, the events from the EAE that have their `startTime` changed, will send out correction messages.

5.2 Optimizations

The above mentioned timestamp correction is not very effective largely due to cascading corrections and also because for every correction message the entire timeline has to rearranged by removing many events and reinserting them. Several optimizations have been implemented along with above scheme and are listed below:

- Overwrite timestamps of old messages: When a new correction message arrives, if there is already a correction for the same destination that is not yet processed, the old message's predicted recvTime value is overwritten. Same scheme works when a correction message arrives earlier than the message itself.

- Use multisend:Many messages destined to different simulated processors but to the same real processor are sent collectively, using a library provided in Charm++.

- Prioritize messages based on the recvTime that they carry. This reduces the number of out-of-order messages drastically.

- Lazy processing: Correction message are processed periodically after certain interval of time. This delay in the processing causes many of the correction messages to be

overwritten. It also amortizes the cost of restructuring the entire timeline over several corrections.

- Batch processing: Many correction messages will be processed collectively. This means that the EAE can be computed for a batch of correction messages and the sequence of operations described in the previous section are done only once for many correction messages.

Even after these optimizations improved the simulation efficiency, simulations of the BG/L were still found to be several times slower when timestamp correction was performed compared to the time taken for simply emulating it. Effect of the cascading corrections and too many out of order messages was still enormous. This can be reduced if execution is not allowed to go far ahead of the correction wave. This is because a large difference between them means there are many wrong events to be corrected. Closely following the execution wave also may not help, as a delayed real message can cause the correction wave to go back. Based on these observations a new gvt-like scheme was developed.

5.3 An approximate GVT scheme

The Global Virtual Time (GVT) is defined as the globally smallest timestamp of all pending messages in the system. All the messages (real and correction) that are currently pending in the system will have a timestamp of no earlier than the current GVT.

We use a heartbeat mechanism to periodically compute the estimated GVT. Every simulated processor of the target machine reports its Simulated Processor Virtual Time (SPVT). This is computed as the minimum timestamp of all the messages in real and correction message queues and messages sent in its timeline during that interval. The timestamp of a message is the predicted `recvTime` that it carries. Every real processor computes the Real Processor Virtual Time (RPVT) as the minimum of SPVTs of all simulated processors on it. The minimum of the RPVTs is value of the estimated GVT (eGVT) to be used for that

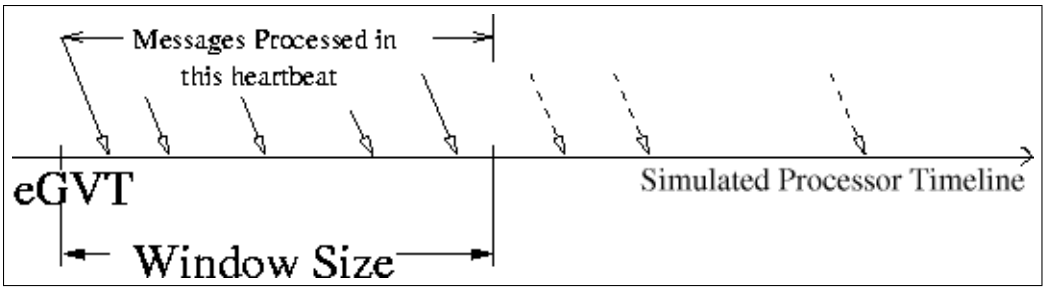

Figure 5.1: Use of eGVT and windowSize at every heartbeat selectively to process messages

interval. Due to the messages in transit that maybe arbitrarily delayed, the value of GVT obtained is just an estimate. Therefore it is not necessary for the new eGVT to be larger than the old value and the reverse may happen in rare cases. However the trend for the eGVT shall always be increasing, thereby carrying the simulation forward. After the eGVT is computed, it is broadcast in the system.

After a new eGVT is obtained, to take advantage of the lazy and batch processing optimizations mentioned earlier, we use a time-window to restrain the advance of the program execution beyond eGVT. The correction messages are buffered and processed periodically at every heartbeat. Only those real messages with a timestamp within the window size from eGVT are executed in the heartbeat interval. This is demonstrated in the Figure 5.1. The time-window advances every time the new eGVT is obtained.

The number of real and correction messages processed is also recorded for every interval. If the number of messages processed is very high, that denotes a high activity period in the simulation. In such a situation eGVT must be updated more often and window advances faster. The heartbeat interval is adaptive and the interval is shrunk in high activity period. Similarly it is expanded when there is a low-activity period. A common deadlock problem in this scheme occurs when the newly computed eGVT does not change when compared to the previous value as no new messages are processed. If deadlock is detected, the window size is increased to allow more messages to be processed.

Chapter 6

Online Parallel Simulation: Results and Case Studies

In his section we present the results of the online parallel simulation. Using our simulator, performance issues of some real world applications on planned machines, specifically on Blue Gene/L can be studied. We describe case studies show that our simulator as a useful tool for performance prediction and performance analysis. It facilitates the development of applications on machines with very large number of processors even before the machine is available.

Furthermore, we are able to do performance analysis for application based on Projections in postmortem fashion. Projections is a performance analysis tool associated with Charm++, it provides the capabilities of detailed event tracing and interactive graphic analysis [3].

For the sake of performance analysis, during the run, events and their details are recorded into applications level logs, one for each simulated processor. It is, however, infeasible to record logs for all processors due to memory and file system constraints. Projections then allows user to choose only a subset of simulated processors to be traced in detail, or trace the whole simulation in less detailed mode - summarizing event data across all processors.

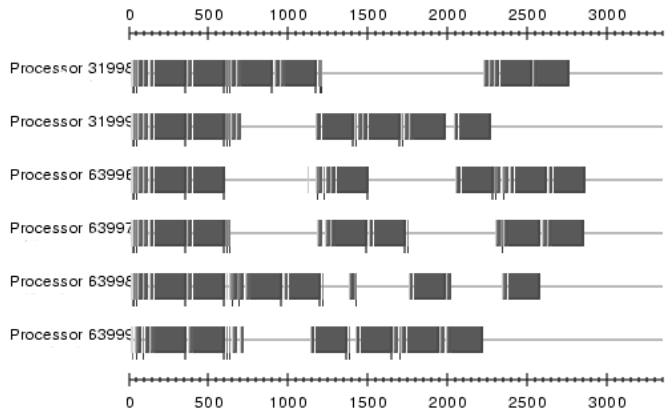

Figure 6.1: Timelines before correction

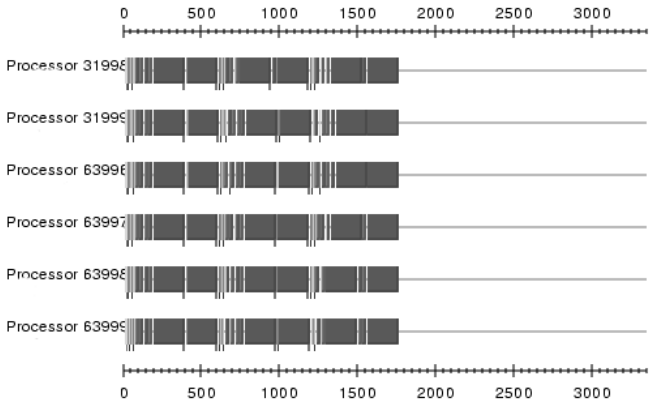

Figure 6.2: Timelines after correction

6.1 Jacobi1D

The previously described Jacobi1D program was used as a case-study to further analyze and validate the behavior of the simulator. The network delay model uses a per-hop and per-corner latency of 5ns and 75ns respectively. The network delay can be increased by scaling both the per-hop and per-corner latency by the same factor.

The timelines of the events on different processors were visualized using Projections. During the run, events and their details are recorded into logs and displayed in the timeline. The timelines shown in figures 6.1 and 6.2 were generated before and after the timestamp correction for a network latency factor of unity. The separation between the events in 6.1 is caused due to the direct or cascaded effect of the out-of-order delivery of messages. A message with a later timestamp can arrive early and thread timer will be advanced to the later timestamp which is wrong. This causes the gaps shown in 6.1. In figure 6.2, the simulator corrects the problem of out-of-order delivery, and the thread timer is correctly updated.

The performance prediction results that we obtained are summarized as follows:

- For a correct timestamp correction scheme we expect same predicted time for the same problem independent of the number of real processors used for simulation. We can use this to test the correctness of the scheme. Predicted performance was indeed found to

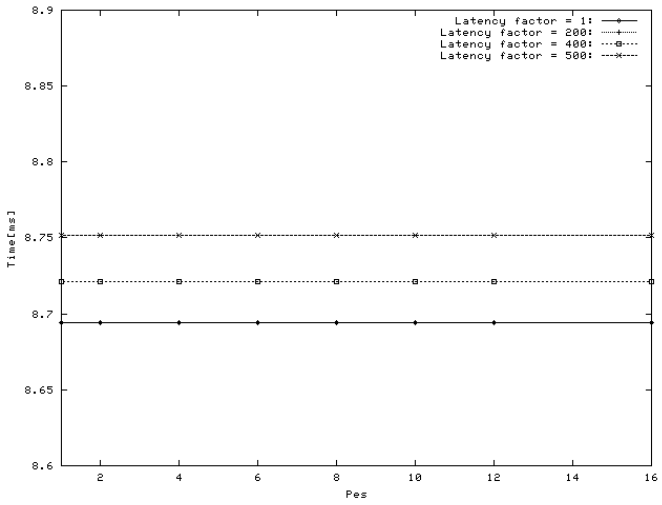

Figure 6.3: Predicted time vs real processors

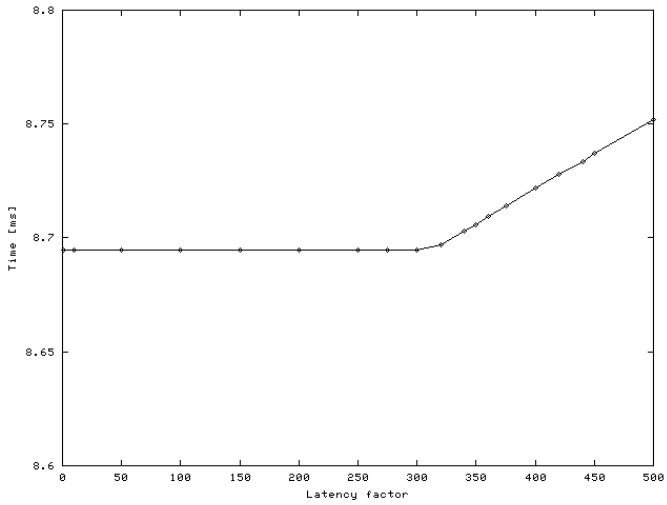

Figure 6.4: Predicted time vs latency factor

be same across different runs and this result for Jacobi1D is shown in Figure 6.3 for different network latencies.

- As we increase the network latency we expect the predicted time to remain constant upto a limit and increase thereafter, due to overlap of computation and communication. The predicted time was measured as function of the factor by which the per-hop and per-corner latencies are increased. The result was as expected and is shown in Figure 6.4. Some amount of delay is tolerated after which, the computational work cannot makeup for the communication delay. So, the predicted time then increases almost linearly.

- The speedup was also measured based on the predicted time for different latency factors as shown in the Figure 6.6. For a very low network latency, the speedup was found to be close to linear, and dropped as the latency factor was raised. For the same amount of work, as the number of simulated processors increases, the work per processor decreases. The computation cannot make up for communication delay in this case and the speedup reduces.

6.2 LeanMD

LeanMD is a molecular dynamics simulation application written in Charm++ and Structured Dagger that was developed at the Parallel Progamming Laboratory, UIUC. It is being developed as the next generation of NAMD [5], a state-of-the-art parallel molecular dynamics applications that is also written in Charm++. However, it is not ready for large parallel machines with hundreds of thousands or even millions of processors due to the limited parallelism exploited in the application. It is essential to develop and experiment new parallelization strategies to effectively distribute work across the extremely large number of processors.

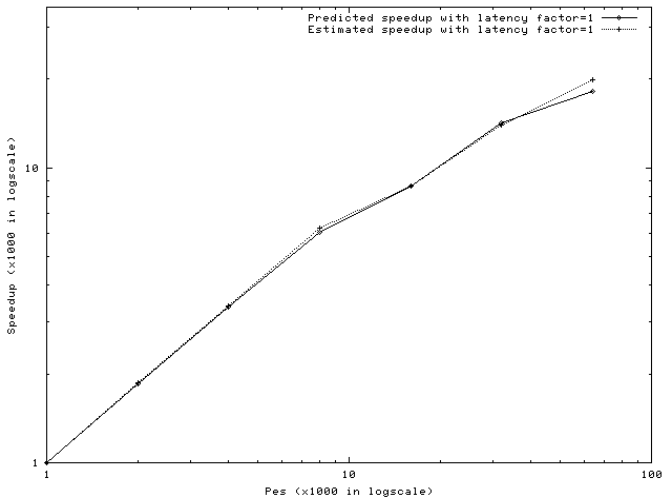

Figure 6.5: Speedup for LeanMD

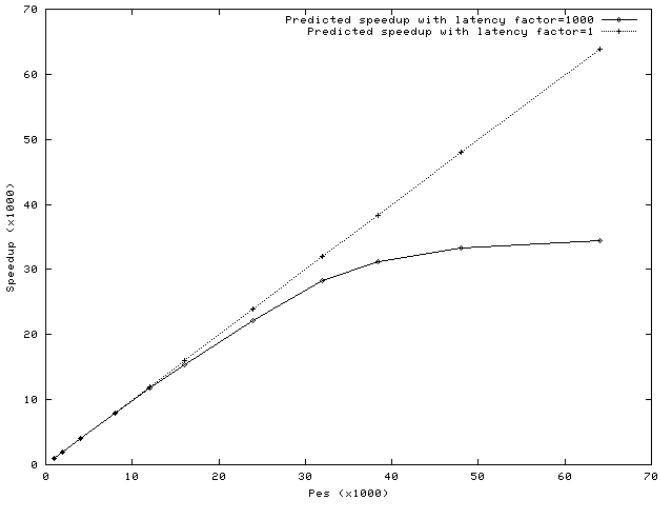

Figure 6.6: Speedup for Jacobi1D

In NAMD, atoms are divided spatially into cells. Interactions between them are calculated every timestep. If the interaction are computed between only the neighboring cells then they are called "one away" interactions. All atoms within a cut-off distance will be accounted for. But this strategy produces a division that is coarsely grained for planned machines such as Blue Gene/C. 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[1] cell-to-cell interactions to compute.

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

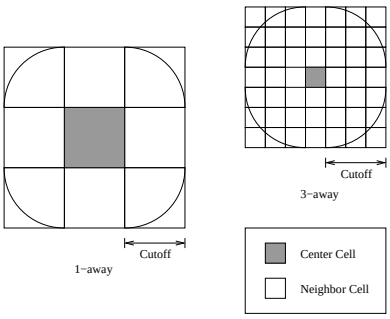

Figure 6.7: 1 away and 3 away cut-off distance

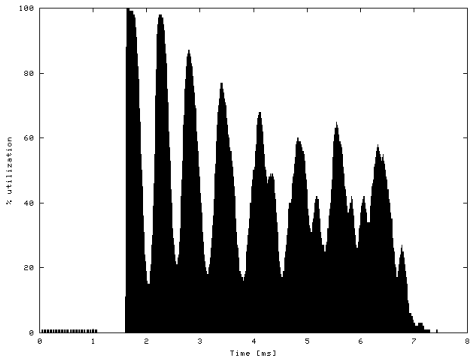

Figure 6.8: Average utilization per interval

The division would lead nodes idle on large machines like BG/L even if each interaction is delegated to a single node.

For creating finer-grained parallelism for cutoff interactions, LeanMD was 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 span the cutoff distance as shown in Figure 6.7. 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 is easily distributed across the 64,000 nodes of the Blue Gene/L.

We have been able to run the LeanMD on our simulator on Lemieux as a real benchmark. We have run 3 sway 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 Blue Gene/L nodes of size from 1K to 64K of full machine size. The predicted speedup is shown in figure 6.5 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 6.8 for 32k simulated processors. The utilization stabilizes at about 50%, but rises and falls within each timestep.

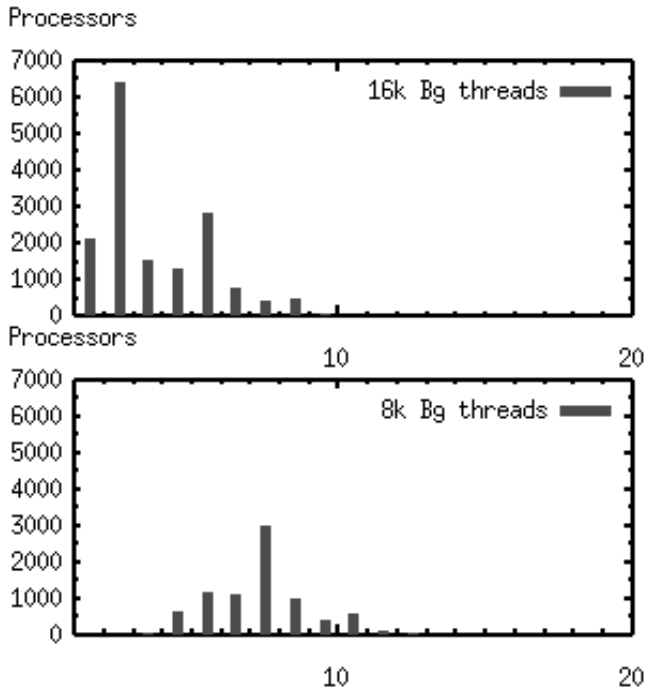

Figure 6.9: Distribution of processors based on load in ms

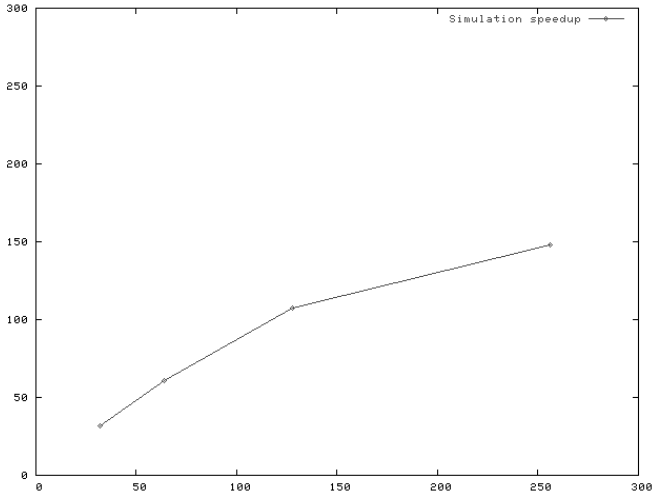

Figure 6.10: Simulation Speedup

This corresponds to the speedup saturation seen in Figure 6.5. To understand the saturation of the speedup we used the performance logs to calculate the load on individual processors. Figure 6.9 shows 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 as high a load as 8ms. This suggests load balance as 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 6.5) 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 overhead or critical paths.

For these simulations we used a no-contention communication model, with possibly too optimistic communication parameters. We plan to 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.

6.3 Performance of the simulation

We measured the performance of the simulator also using leanMD as a sample application. We demonstrate the scalability of the parallel simulator in Figure 6.10. The simulation was found to scale reasonably over hundreds of processors. The efficiency of the simulation depends on the number of correction messages sent. In one simulation, correction and real messages sent were compared for different simulated processors as shown in Table 6.1. 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	20,040,000	20,180,000	20,420,000	20,930,000
Corr. Msgs	357,351	305,487	126,629	59,762

Table 6.1: Proportion of correction messages

Chapter 7

Performance Prediction Using Post-Mortem Analysis

This section explores the post-mortem analysis for performance prediction. In this case, there is no timestamp correction during the actual run of the parallel program and hence no correction messages have to be handled. This approach however uses the event dependencies created using Structured Dagger to determine the causality of the events when the timeline is corrected after the simulation finishes. The simplicity of this approach compared to an online correction makes it attractive.

7.1 Sequential approach

An alternative to doing parallel discrete event simulation is to let the program run without any corrections in the message delivery order. We record traces during emulation, and then run a sequential trace-driven simulation. The simulation checks that the messages are always processed in the order of their timestamps. This is possible because the least timestamp to be processed is always globally available from the traces.

In this approach, first the program is run on the emulator without any modifications. Event logs are created for every entry method and Structured Dagger *atomic* executions. The dependencies between different methods is captured creating a sequence of backward and forward dependents for each event and written into the log. Each message sent by an

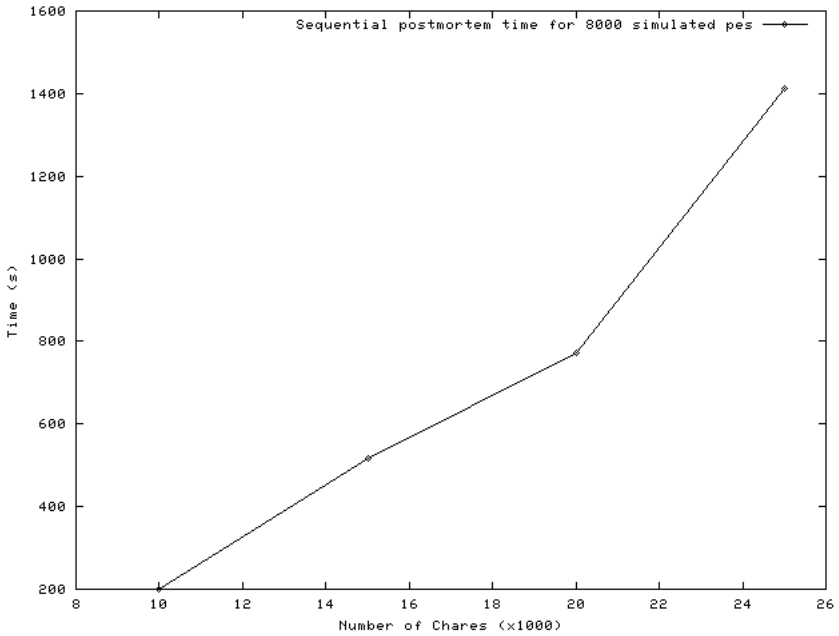

Figure 7.1: Sequential post-mortem times for different chares on 8000 simulated processors

event is also logged as the receive time of that message may have to be corrected when the events starting time is changed. The links to the forward and backward dependents are also logged.

When the logs are read, the timelines are reconstructed in the correct order. Every time the event with the globally least effRecvTime event is selected to be added to its corresponding timeline. The very first event that starts the program is always inserted first as it has a zero effRecvTime. From then on for every event E that is added to the timeline, all the events that received messages from E recvTimes updated. Also, if the recvTime of an event changes, the effRecvTime of the forward dependents is recomputed.

7.1.1 Results of the sequential approach

The correctness of the sequential approach was confirmed by comparing the resulting timelines with the online correction scheme. The time for the sequential post-mortem simulation was measured for different problem sizes. A simple jacobi 1D program with only 1 iteration was run for only 8000 simulated bluegene processors. However, the number of Chares

(concurrent Charm++ objects) each carrying equal amount of work was varied. The result is of the sequential times is shown in 7.1. The large simulation times make it impossible to simulate a large problem for a full size BG/L. For larger simulations, the memory consumption is expected to increase and the performance may worsen due to swapping of data from memory to disk. This shows that it is necessary to develop a parallel approach for the same. Sequential approach for post-mortem correction though simple, is useful only for smaller sized problems and simulated machines.

7.2 Parallel approach using POSE

POSE [8] is a parallel discrete event simulation environment developed at the Parallel Programming Laboratory, UIUC. It is built on top of Charm++ and is broadly applicable as it supports many synchronization protocols and speculation strategies. POSE was designed for scalability of parallel and distributed simulations. It has an object oriented language built over Charm++ for modeling complex discrete event systems.

For the use of POSE, the logs are to be read and processed in parallel. For every simulated processor, the index of the file containing the logs of its timeline is computed. The files have headers for describing the offsets of various timelines that they contain. This simultaneous reading of the logs reduces the initialization overhead.

The events are triggered using the message-sent information and the dependency just as in the sequential method. But in this approach each simulated processor is represented by a Poser. Posers are objects similar to Chares in Charm++ and execute events in timestamp order without violating the causality order. The number of physical processors used for the processing of logs can be different from the number used for running the simulation that generated those traces.

The logs are read from the traces. Each simulated processor has the list of all the tasks that it executed. The events are invoked using a POSE_invoke construct that takes an event

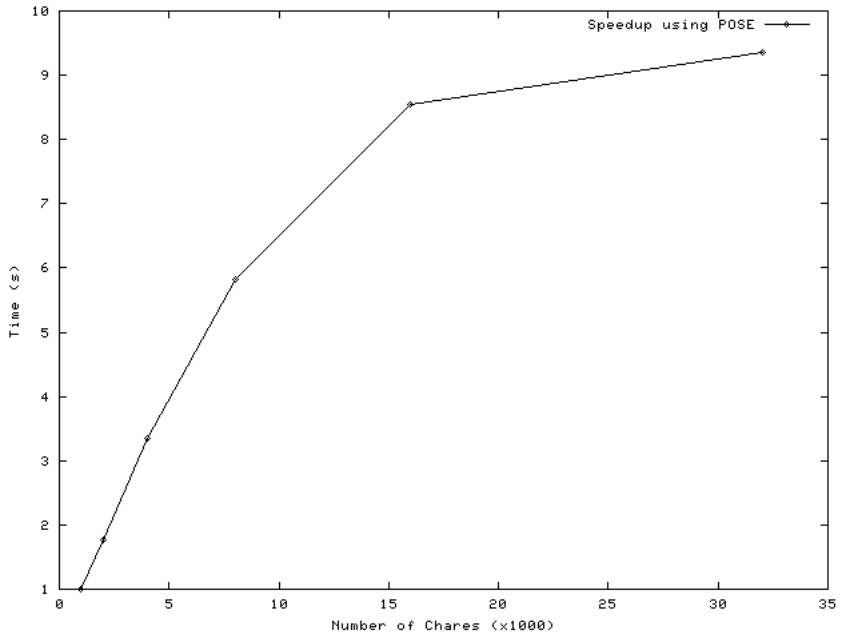

Figure 7.2: Speedup using adaptive strategy in POSE, for 8000 simulated BG/L processors

method and a time-offset after which it has to be invoked. For events triggered by messages the time-offset would by the predicted network latency. For a Structured Dagger event when all of its backward dependents are ready, it is invoked at the end of the current event. The execution time can be elapsed using the elapse construct in POSE.

7.2.1 Results using POSE

The parallel post-mortem analysis was performed for a Jacobi1D on 40000 chares simulated over 8000 simulated processors using adaptive strategy in POSE. The speedup of the simulation is presented in Figure 7.2.

7.3 Post-mortem vs online approach

In both the post-mortem approaches described above, the logs files have to be generated and written to the disk. In the actual implementation a log file is created for every physical processor that holds the details about the events in all the timelines of all the logical processors

on that physical machine. An online correction scheme however does not require that the simulation details be recorded and the predicted time can be obtained immediately at the end of the simulation.

The post mortem approach however offers features that the online approach cannot provide. In the post-mortem approach it is possible to rerun the simulation from the same traces with varying network parameters (by changing the `recvTimes` of the messages) or try different data processing speeds (by varying the execution times). A more detailed analysis can be performed if the data recorded in the traces is made more fine-grained, for example by describing the number of floating-point or integer operations for the execution of every log. In this case predicted performance can be measured as function of the performance of floating-point units.

Chapter 8

Conclusion

We have implemented a simulator that can make predictions of the performance of any application on parallel machines that are yet to be built. We have also demonstrated its use on application benchmarks for the specific case of Bluegene/L. The ideas presented in this thesis are steps in the direction of developing an accurate and low-overhead simulator for large parallel machines. It can be used for identifying applications that suit the certain architectures and evaluating their performance on several target machines.

8.1 Future work

The future work would involve making the simulator more closer to the real target machine itself. This work involves improving the estimates of the network overhead and the execution time of code blocks. Instead of using a heuristic value as in the current simulator, realistic network effects can be studied by building an exact network simulation using POSE. This would model network-link contention, queuing delays and get a more real estimate of the network latency. A cycle accurate simulator for the machine depending on its architecture can provide better estimates of the sequential code blocks than using user-defined estimates as in the current simulator. The post-mortem analysis can be made more flexible by changing the execution times and receive times of different events after the traces are read, so that the effect of different design parameters of the machine can be studied.

References

[1] L. V. Kale and Milind Bhandarkar. Structured Dagger: A Coordination Language for Message-Driven Programming. In *Proceedings of 2nd International Euro-Par Conference*, volume 1123-1124 of *Lecture Notes in Computer Science*, pages 646–653, September 1996.

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

[3] Laxmikant V. Kalé, Sameer Kumar, Gengbin Zheng, and Chee Wai Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop (ICCS'03)*, Melbourne, Australia, June 2003.

[4] Robert Brunner L. V. Kale, Milind Bhandarkar and Joshua Yelon. Multiparadigm, Multilingual Interoperability: Experience with Converse. In *Proceedings of 2nd Workshop on Runtime Systems for Parallel Programming (RTSPP) Orlando, Florida - USA*, Lecture Notes in Computer Science, March 1998.

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

[6] Neelam Saboo and L. V. Kalé. Improving paging performance with object prefetching. Technical Report 01-02, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, July 2001.

[7] Neelam Saboo, Arun Kumar Singla, Joshua Mostkoff Unger, and L. V. Kalé. Emulating petaflops machines and blue gene. In *Workshop on Massively Parallel Processing (IPDPS'01)*, San Francisco, CA, April 2001.

[8] Terry Wilmarth and L. V. Kalé. POSE: A study in scalable parallel discrete event simulation. Technical Report 03-01, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, June 2003.

[9] Gengbin Zheng, Gunavardhan Kakulapati, and L. V. Kale. A parallel simulator for performance prediction of large parallel machines. Technical Report 03-02, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, June 2003.