Simulation-Based Performance Prediction for Large Parallel Machines

Gengbin Zheng, Terry Wilmarth, Praveen Jagadishprasad, Laxmikant V. Kalé
Dept. of Computer Science
University of Illinois at Urbana-Champaign
{gzheng, wilmarth, jagadish, kale}@cs.uiuc.edu

Abstract

We present a performance prediction environment for large scale computers such as the Blue Gene machine. It consists of a parallel simulator, BigSim, for predicting performance of machines with a very large number of processors, and BigNetSim, which incorporates a pluggable module of a detailed contention-based network model. The simulators provide the ability to make performance predictions for very large machines such as Blue Gene/L. We illustrate the utility of our simulators using validation and prediction studies of several applications using smaller numbers of processors for simulations.

1 Introduction

Parallel machines with enormous compute power and scale are now being built consisting of tens of thousands of processors and capable of achieving hundreds of teraflops of peak speed. For example, the Blue Gene (BG/L) machine being developed by IBM and slated for early 2005 delivery, will have 128K processors and 360 teraflops peak performance. Ambitious projects in computational modeling for science and engineering are gearing up to exploit this power to achieve breakthroughs in areas such as rational drug design, genomics, proteomics, engineering design and computational astronomy.

Development of a programming environment for such machines is a significant challenge. It may require qualitative changes to the way we write parallel programs in order to exploit the enormous compute power available. Further, it is also important to understand performance issues in specific algorithms thoroughly, so that next-generation applications can be written to scale to such large machines. We have been engaged in a project [1] to address these challenges for over two years. In this paper we summarize our progress and findings so far.

We explored CHARM++ and Adaptive MPI as an appropriate programming model for large machines because of its ability to virtualize processors [7], allowing programmers to not worry about specific actions running on specific processors. This property seems essential for dealing with large machines, because it would be impractical to think about what is running where on 100K processors. The rest of the paper is organized as follows. We first use an emulator to explore scaling CHARM++ and Adaptive MPI [3] to run on large machines. Next in Section 3 we present our performance prediction system, based on parallel discrete event simulation techniques, and some novel ideas to avoid re-execution during optimistic simulation. We present recent performance results using the simulator for structural dynamics computations involving the Finite Element Method (and unstructured grids) in Section 4 and Molecular Dynamics Simulation. An overview of future and ongoing re-

[1] This work was supported by a National Science Foundation grant NSF NGS #0103645.

search issues is presented in the final section.

2 Emulating Petaflops Machines

Deciding the characteristics of an ideal programming environment for a massively parallel machine like IBM Blue Gene is a challenging task. This is because dealing with tens of thousands or even millions of processors requires a qualitative change in both the programming environment and the runtime system. Further, it is very challenging to evaluate these programming models in a real context before such machines are built.

To this end, we have developed a software emulator to mimic a class of target parallel machines, on which a multi-tier programming model is built. The lowest layer, a low level programming API enabled by the emulator provides a general message passing interface for implementing a high level parallel language which forms a middle layer in our programming environment. The higher level components in the programming environment consist of domain specific languages and libraries.

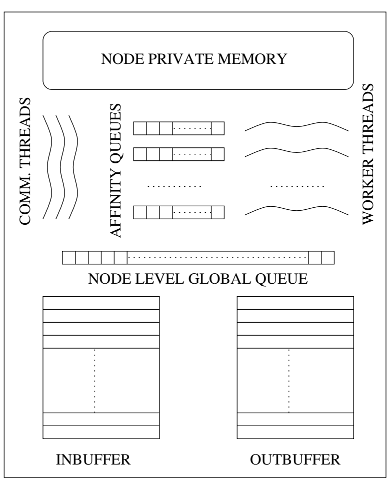

Figure 1. Functional view of an emulated node

The lowest level model strives to provide access to a machine's capabilities. In the programmer's view, each node consists of a number of hardware-supported threads with common shared memory (see Figure 1). Within a node, we divide threads into *worker* threads and *communication* threads. A runtime library call allows a thread to send a short message to a destination node. The header of each message encodes a handler function to be invoked at the destination. Communication threads check incoming messages from the *InBuffer* and put the messages in either a worker thread's *affinity* queue or a node level global queue. Worker threads repeatedly retrieve messages from both queues and execute the handler functions associated with the messages. We believe this low level abstraction of the petaflops architecture is general enough to encompass a wide variety of parallel machines with different numbers of processors and co-processors on each node. The details of the emulator and its API were presented in [11].

Despite its generality, programming in this low-level message-passing model is difficult. The programmer must decide which computations to run on which node. The programming environment at the higher level relieves the application programmer of the burden of deciding where the subcomputations run.

We evaluated CHARM++ as a parallel programming language for petaflops machines, as well as the popular MPI methodology. CHARM++ is an object-based portable parallel programming language that embodies message-driven execution. A CHARM++ program consists of parallel objects and object arrays[8], which communicate via asynchronous method invocations. CHARM++ includes a powerful runtime system that supports automatic load balancing based on migratable objects. CHARM++ has been ported to the emulator as reported in [14].

Adaptive MPI, or AMPI, is an MPI implementation and extension based on CHARM++'s message-driven system, that supports processor virtualization[7]. AMPI implements virtual MPI processes (VPs) by migratable user-level threads, several of which may be mapped to a single physical processor. Taking advantage of CHARM++'s load balancing framework, AMPI supports adaptive load balancing by migrating MPI threads. In this environment, MPI is a special case of AMPI when exactly one VP is mapped to a physical processor.

3 Performance Modeling Environment

Accurately estimating the performance of target applications on massively parallel machines is useful to application programmers in adapting their codes to the new architectures. Such a performance estimator is also an essential tool for designers of petaflops machines who, in order to make good design choices, need to evaluate alternate architectural features in the context of specific benchmarks.

It is clearly impractical, if not impossible, to simulate a million processor machine on a single processor. Instead, we aim at the challenges involved in carrying out such simulations on a conventional parallel machine with over 1,000 processors, attaining the desired timing accuracy using multi-level simulation techniques.

For this purpose, we have developed a performance modeling environment which consists of *BigSim* simulator [13] for performance prediction of large parallel machines.

In the rest of this section, we will present the simulation techniques and optimizations we explored in the performance prediction of parallel applications with simulations using different degrees of fidelity.

3.1 Parallel Discrete Event Simulation

All of the important behaviors that model a parallel application on a very large parallel machine can be efficiently described as actions occurring at a particular time and lasting for a known duration. These behaviors are thus best simulated with a parallel discrete event model.

However, performance prediction for parallel applications with *parallel discrete event simulation* (PDES) is very challenging due to complexity of the communication system and non-determinacy of the simulation. Messages may arrive out of order, arising from the fact that we are using multiple processors to carry out the simulation. As a result, messages with later time stamps may arrive before messages with earlier timestamps, resulting in causality errors and destroying the accuracy of the simulation.

Traditional methods in PDES of correcting this involve high synchronization overheads. For example, in optimistic concurrency control these overheads include: (a) checkpointing overhead, (b) rollback overhead and (c) forward reexecution overhead. These overheads may be prohibitive given the size of the simulated petaflops machines.

One simple observation of parallel applications leads to optimizations in PDES to make such simulations feasible. That is the *inherent determinacy* often found in parallel applications. 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 differ slightly, they carry out the same computations. Our approach takes advantage of this characteristic of parallel applications to improve the simulation efficiency.

3.2 Component Performance Models

The BigSim simulator is an extension to the emulator described in the previous section. Converting the emulator to a simulator requires correct estimation of the time taken for sequential code blocks and for network messaging. We have adopted a range of possible methods for prediction with different degrees of accuracy.

To predict the computation time for a target machine (which may not exist), we use several heuristic approaches as described below to estimate the CPU time. They are listed in the increasing order of accuracy and the complexity involved.

1. User-supplied expression for every block of sequential code estimating the time that it takes to run on the target machine. This is a simple and highly flexible approach. However it burdens the user with the onus of accurate estimation.

2. Wallclock measurement of the time taken on the simulating machine can be used via a suitable multiplier (scaling 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 interpolation approach using the time for each of these operations on the target machine to estimate the total computation time. Cache performance and memory footprint effects can be approximated by the percentage of memory accesses and the 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 for the target machine. However, instruction level simulation is very expensive.

The first three of the above described methods are currently supported in BigSim simulator, while we plan to support the last in future.

To simulate the network environment of the target machine for message passing, the following approaches are explored:

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

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

3.3 Online Simulation: BigSim

BigSim is based on direct execution of an application on the emulator described previously. It represents the network as a three dimensional torus network with latency-based modeling. To deal with causality errors, BigSim lets the emulated execution of the program proceed as usual, while concurrently running a parallel algorithm that corrects timestamps of individual messages. This *online* mode of simulation provides several advantages. For example, it makes it simple to simulate dynamic characteristics of a parallel application such as dynamic adaptive load balancing.

Online mode BigSim simulation has been shown to be very efficient and is capable of performing simulations with very large configurations [13]. It applies to simulations of a large category of applications that do not require high fidelity in network modeling.

There are a few drawbacks to the online simulation mode. First, if we choose to vary the latency in the model, we must re-run the entire simulation. Second, we cannot efficiently model the complexities of a contention-based network model simultaneously with execution of sequential computation blocks. However, for the category of applications we are addressing, execution behavior of the sequential computation blocks does not change. For this reason, we can use the emulation mode to generate logs containing this information, and then rely on postmortem simulation to predict the behavior of the application on a variety of network configurations. We discuss this postmortem simulation mode in the next section.

3.4 Postmortem Simulation

In postmortem mode, the BigSim emulator generates logs of sequential computation blocks, the messages generated by each of these blocks, their send time relative to the beginning of the block, and the dependencies between these blocks. This information is sufficient to produce detailed simulations of an application's behavior on a variety of network topologies and contention models.

This type of simulation is more complex and requires the flexibility of a full discrete event simulation model. It also presents the problem of fine granularity of computation. Since most of the interactions modeled involve the transmission of packets through the network model, actual computation time on each modeled entity is minimal. This presents challenges to the parallel simulation of these models since the overhead of event synchronization is high relative to the computation time. We make use of POSE[12], a parallel object-based simulation environment specifically designed to handle such simulations, both sequentially and in parallel.

Postmortem simulation can operate in two modes. In the first mode, a simple latency-based network model is used to correct the start time on each sequential computation block. This is described in more detail in Section 3.4.2. The second mode uses a detailed contention-based network model to perform the start time corrections. We discuss this further in Section 3.4.3.

3.4.1 POSE

POSE is built in CHARM++ which supports the *virtualization* programming model, an approach we believe will give rise to great improvements in PDES performance [7]. The logical processes (LPs) of PDES can be mapped onto CHARM++'s *chares* (parallel objects) in a straightforward manner. Similarly, we use timestamps on messages as priorities and thus the CHARM++ prioritized scheduler takes the place of an event list. Virtualization provides the simulation programmer with a view of the program consisting of the entities in the model and not the underlying parallel configuration.

In POSE, simulation entities, or *posers* are special types of chares that have a data field for *object virtual time* (OVT). This is the number of simulated time units that have passed since the start of the simulation relative to the object. Posers also have *event methods* similar to CHARM++ *entry methods* (invoked by sending messages from one object to another, possibly on a different processor), with the main difference being the presence of a data field for *timestamp* in all messages sent to invoke an event method.

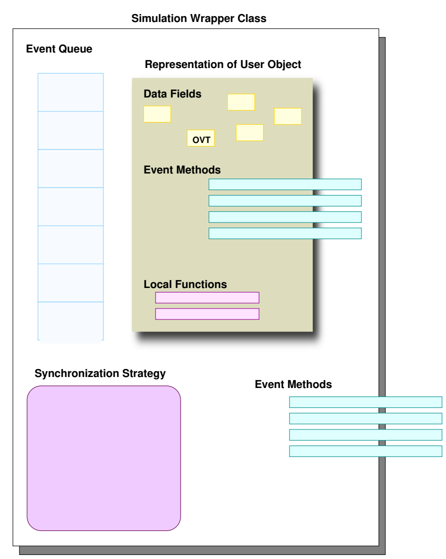

Figure 2. Components of a poser.

Posers can pass simulated time in two ways. First, they can *elapse* a certain amount of local time (presumably performing some activity). This advances the OVT of the poser. Second, an *offset* can be added to event invocations. This is used to schedule a future activity or to indicate *tran-*

sit time in a simulation. For example, suppose the event being invoked involves the movement of data such as a packet being sent over a network, and it takes t time units to transmit it; we would schedule an event at the point receiving the packet at a time that is offset by t from the current time.

Each poser has its own event queue and an instance of a synchronization strategy. Its internal state is encapsulated in an object that receives event messages and queues them locally for execution on the internal state. Figure 2 illustrates this structure. Thus, the scope of simulation overhead resulting from a synchronization error is limited to the entity on which the error occurs. Since different entities may have different behaviors, this limits the effects of those behaviors to a smaller scope, and allows the synchronization strategy to adapt to the behavior of the object.

Speculative Synchronization POSE makes use of optimistic concurrency control as in TimeWarp[4]. When an object receives an event, it gets control of the processor. The object's synchronization strategy is then invoked and checks for any synchronization error corrections (rollbacks, cancellations) before it performs forward execution steps (events).

Here the opportunity to perform *speculative computation*[5] arises. All optimistic strategies perform some amount of speculative computation. In more traditional approaches, an event arrives and is sorted into the event list and the earliest event is executed. We know the event is the earliest available on the processor, but we do not know if it is the earliest in the entire simulation, thus executing it is speculative.

In our approach, we have a *speculative window* that governs how far into the future beyond the global virtual time (GVT) estimate an object may proceed. Speculative windows are similar to the time windows of other optimistic variants, except in how events within the window are processed. Events are inserted into the event queue on the object for which they are destined. In POSE, when the object invokes the synchronization strategy to process events, *all* events with timestamp $t \geq$GVT within the speculative window are executed. The later events are probably not the earliest in the simulation, and it is likely that they are not even the earliest on that processor. We allow the strategy to speculate that those events are the earliest that the *object* will receive. By handling events in bunches, we reduce scheduling and context switching overhead and benefit from a warmed cache, but risk additional rollback overhead.

POSE has an adaptive synchronization strategy that strives to execute more events at a time while minimizing rollbacks. This strategy automatically adjusts the size of an object's speculative window according to the rollback history and queued future events on the object. This maximizes the effects discussed earlier, localizing the effects of individual object's behaviors.

3.4.2 Simple Latency-Based Network Simulation

For the simple latency-based network simulation, we read the log files generated by the BigSim emulator. The size of the log is proportional to the number of messages exchanged. An application execution was emulated on some configuration, and all the sequential computation blocks (called *tasks* in postmortem simulation), messages generated by these tasks, and the dependencies between tasks are recorded in these logs. In our simulation, we recreate entities in POSE to model the processors and nodes of the emulation. We then read in the tasks and use the simulation to pretend to execute them. For each task, we know what it depends on, what depends on it, the duration of the task, and what other tasks were generated by it and when these other tasks were generated (as an offset from the current task's start time). We also have an estimate of network la-

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

Figure 3. Feigned task execution in POSE.

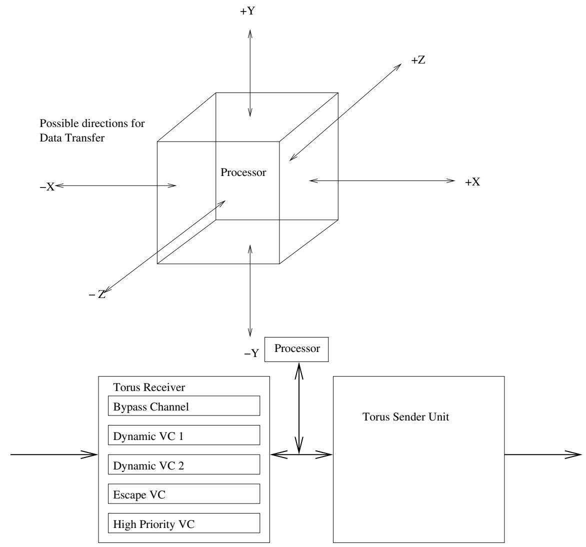

Figure 4. Processor sending and receiving messages in all 6 directions through sender and receiver units

tency which we use to determine how much time generated tasks spend in transit to the processor on which they will be executed.

What we do not know is exactly when each task started (though we do have an uncorrected timestamp for each task), and without that information, we do not know how the emulated application performed. Given the information above, we start the first task off at virtual time zero, and let the tasks "execute" and record the virtual time at which each task starts. The algorithm for this feigned execution is shown in Figure 3.

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

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

When all tasks have been executed, they should have correct timestamps and the final global virtual time (GVT) should represent a correct runtime for the emulated application. Section 5 shows the performance of this postmortem simple latency-based network model.

3.4.3 BigNetSim: Detailed Contention-based Network Simulation

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

In BigNetSim, there are posers corresponding to processor, switch (including ports and virtual channels), channels and network/processor interface. Other non-tangible network entities like protocol stack, flow control, routing, arbitration, topology, etc. are modeled in event methods across one or more posers. There is a network configuration file which includes parameters for bandwidth, latency, ports, virtual channels, routing scheme, buffer size, packet size and option to print link contention statistics.

The network simulator framework is flexible to model arbitrary topology, routing algorithm, Input and Output Virtual Channels (VC) Selection policies etc. For this paper, we chose a network design close to the actual Blue Gene/L network [9]. The processor interface consists of network injection and reception FIFOs for transferring messages. There are sender and receiver units in each node which send and receive messages to and from the network. There are internal channels connecting the receivers and senders in the same node, and external channels connecting neighboring senders and receivers.

Messages are split into packets of up to 256 bytes and injected into the network. The receiver then sends out an arbitration request to the sender units seeking to transfer data. Each receiver has four VCs as shown in Figure 4. Escape VC helps in preventing deadlocks. Bypass channel can be used to flow through a node without any buffering. Each buffer has 1KB of memory, enough to hold four full sized packets. Escape VC can be used only when dynamic VCs are unavailable. Escape VC can be used only if we can guarantee that space required for a full sized packet i.e 256B is available, even after reserving space for the current packet.

Routing can be done in any of the six directions. The simulator offers both static and adaptive routing. Both routing approaches select shortest distances based on the torus topology. Static routing routes the packet fully in X dimension, then in Y and Z dimensions respectively. Adaptive routing routes the packet on the least loaded virtual channel. "Join the Shortest Queue" (JSQ) algorithm is used to select the VC with maximum available buffer space among the valid directions. "Serve the Longest Queue" (SLQ) strategy is used to determine which receiver VC wins the right to request senders. With this strategy, the VC with least available buffer space is selected. Virtual Cut-Through buffering is modeled, ensuring reduced latency by pipelining routing and arbitration. Only the route flits endure routing delays and the data flits have a free flow.

Output contention in a switch due to simultaneous access for a sender from multiple receivers is modeled. Other types of contention modeled include input contention in a switch due to Head-Of-Line blocking and contention in the processor interface due to multiple message transfers.

BigNetSim network simulator is very communication intensive when run in parallel with applications like NAMD. In a typical simulation it transfers millions of small message packets.

4 Applications and Case Studies

Parallel algorithms developed for conventional parallel machines are not necessarily appropriate or efficient for petaflops machines. Parallel algorithms for such class of petaflops machines must handle low bisection bandwidth and relatively low memory-to-processor ratio. They must exploit the availability of dedicated communication threads and the existence of multiple parallel communication links. They must also meet the challenges of scalability.

We have developed and evaluated several parallel frameworks and their applications for petaflops class machines including Molecular Dynamics (MD) and FEM framework.

The Molecular Dynamics simulation of biomolecules is one of the important applications for Blue Gene/L and other large parallel machines. NAMD [6], recipient of a 2002 Gordon Bell Award, is a production quality parallel molecular dynamics code designed for high-performance simulation of large biomolecular systems. Based on Charm++ parallel objects, NAMD scales to thousands of processors on high-end parallel platforms [10] and tens of processors on commodity clusters using switched fast ethernet.

The Finite Element Method (FEM) is a popular technique often used in the study of fracture and structural mechanics. We have developed a parallel framework [1], called the CHARM++ FEM Framework to make it easy to parallelize a serial FEM code. The framework handles the finite element mesh that discretizes the problem domain, partitioning the mesh for parallel execution, and providing easy to use communication primitives defined on the mesh.

In this section, we focus on these two applications to demonstrate the utility of BigSim simulators.

4.1 Exposing Unexpected Obstacles

Our runs with BigSim exposed a number of unexpected bottlenecks and limitations to scalability of FEM applications. First, large meshes must be generated; this is difficult with today's tools. Second, the meshes must be partitioned for parallel execution.

One difficulty is that real problems are defined on complicated domains, like machine parts and fracture surfaces, so generating a mesh for the domain is a nontrivial task. Meshes are usually generated by special meshing software in an offline, serial process, so no publicly available meshing software can generate billion-element meshes. A typical solution to this is to first generate a relatively coarse mesh in serial to capture the basic geometry of the domain, then use parallel mesh refinement (or mesh multiplication) to generate more elements where needed. The FEM framework does not handle mesh generation, but it includes rudimentary capabilities for parallel mesh refinement.

Once a mesh is generated, it must be partitioned, and the pieces sent to different processors for parallel execution. The FEM framework initially used the serial Metis partitioning library, so the partitioning was performed completely on one processor, which was a bottleneck for large meshes running on BigSim. For example, Metis library consumes memory proportional to the number of output pieces, not the total size of the mesh; so even our 4GB machine ran out of memory when partitioning a relatively small 5M element mesh into more than 16K pieces. Even on a machine with much larger memory, we found that Metis is still not able to partition the mesh to more than 32K pieces due to the floating point and integer overflow. One solution is integrating the parallel ParMetis partitioning package to avoid the serial mesh partitioning bottleneck, which allows us to use larger meshes and scalably partition the mesh. An alternative approach is to use a simpler but inaccurate mesh partitioner such as geometric recursive coordinate bisection, then fix the resulting load imbalance using our load balancing framework.

4.2 FEM Grainsize on Large Machines

FEM computations have a characteristic parallel communication pattern—each processor first exchanges data with neighboring processors, then performs local computation and repeats the process. The time spent in local computation in each step can be quite small, especially for larger machines and smaller meshes. This small "grainsize" means communication happens more often, which can lead to poor performance.

For example, with a 1M element mesh running on 100,000 processors, each processor might only have 10us of computation between messaging phases. Since message latencies are typically of the order of several micro seconds, processors will spend all their time communicating and efficiency will be very low.

Communication latency can be hidden to a large extent with the technique of "**processor virtualization**", in which the problem is decomposed into more pieces than processors, and the pieces scheduled dynamically based on which messages are available. CHARM++ and the FEM framework fully support virtualization, and in fact require no extra user code for a virtualized run.

Another complementary approach to handle communication latency is the ghost cell expansion method [2], where redundant computations around each processor's border are used to decrease the frequency of message exchange. This multiple-ghost approach has only been implemented for structured grids, however, and the extension to unstructured grids, while conceptually straightforward, would be complicated to implement.

4.3 NAMD Simulation and its Validation

We have compared the actual execution time of NAMD with our simulation of it using BigSim on Lemieux. Lemieux is a terascale computing system comprising an HP alphaserver cluster with a Quadrics interconnection network. As a benchmark system we used Apo-Lipoprotein A1 with 92K atom. A multiple time-stepping scheme with PME (Particle Mesh Ewald) involving a 3D FFT every four steps is performed. The result is shown in Table 1. The first row shows the actual running time of NAMD on 128 to 1024 processors on Lemieux; the second row shows the predicted running time using BigNetSim offline simulation on a Linux cluster. The network parameters are based on Quadrics network specifications. It shows that the simulated execution time is close to the actual execution time.

Processors	128	256	512	1024
Actual run time (ms)	71.5	40.3	23.9	17.6
Predicted time (ms)	75.8	43.6	25.1	20.8

Table 1. Actual vs. predicted time

4.4 NAMD Communication Pattern Analysis

Network statistics like link utilization and contention obtained from the network simulation, can be used to visualize the communication pattern of the application. This helps to identify communication bottlenecks in the applications for performance optimizations.

We use NAMD as a case study to illustrate this utility. In NAMD, the atoms are divided spatially into cells roughly the size of cutoff distance. Local interactions are calculated each timestep between only the nearest neighbor cells. Each simulation timestep starts with multicast communication for cells to send the atom data to the nearest neighbors; the computation begins after that, followed by communication that sends the force result back to the cells. Note that due to the latency tolerance in CHARM++, communication and computation can overlap.

Figure 5 shows the average link utilization during the whole runtime in a 128 node 15 time step NAMD simulation. The irregular utilization pattern matches with bursts of traffic during time step boundaries as shown in Figure 6. From the magnified view of a timestep in Figure 7, we can see most communication happens at the beginning and end of the timestep as expected, while overlapping of communication and computation can be observed.

The first synchronization is a simple barrier at around the 1200^{th} Interval and the second one is a load balancing step between 1285^{th} and 1315^{th} Interval. The load balancing involves a collection followed by broadcast operation. The ensuing traffic results in huge contention as shown in Figure 8. Figure 9 shows the number of links where utiliza-

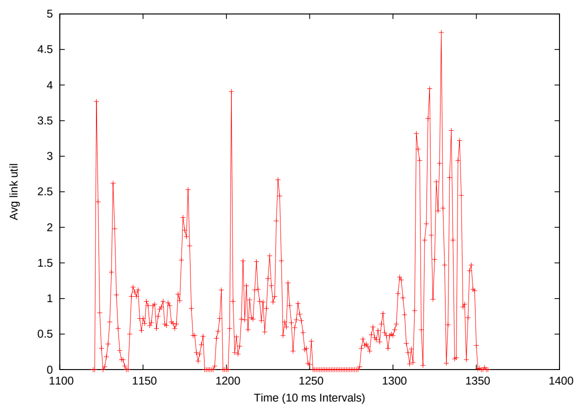

Figure 5. Average Link Utilization

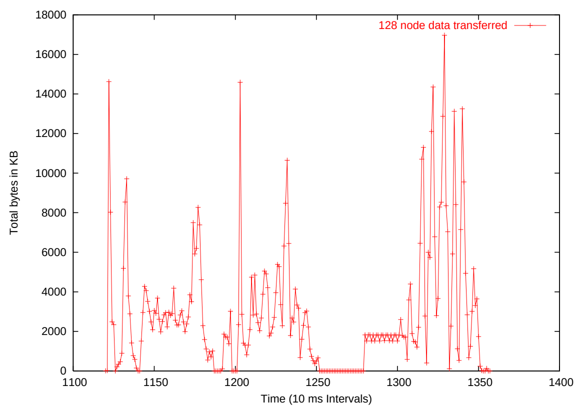

Figure 6. Data Transferred (KB) during Full Simulation

tion is greater than 30 percent in an Interval. We can clearly see that many links during load balancing have utilization greater than 30 percent. This is significant as a typical message is transferred in a few microseconds and it takes many messages to be transferred in 10 milliseconds to have a high link utilization. Blue Gene/L has a separate tree network for doing collective operations, which is not modeled in the current network simulator. We think that it will help alleviate the network contention of such broadcasts.

4.5 FEM Scalability Study

We studied the performance of a CHARM++ FEM Framework program, which performs a simple 2D structural simulation on an unstructured triangle mesh. We chose a relatively small problem with a 5 million element mesh, so

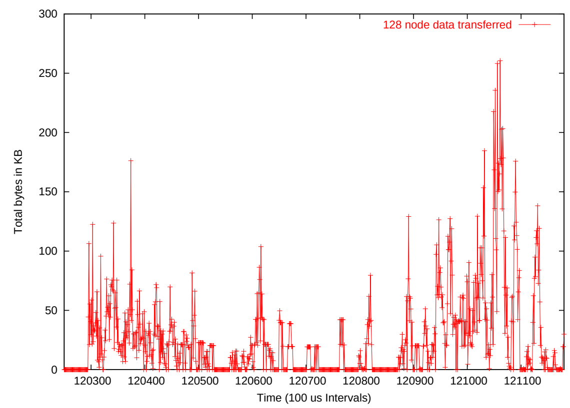

Figure 7. Data Transferred (KB) in a Single Time Step

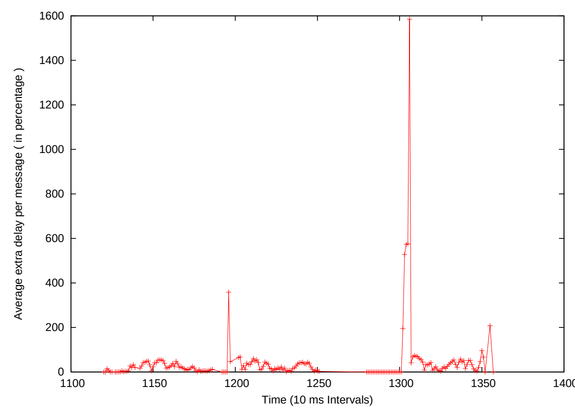

Figure 8. Contention encountered by messages

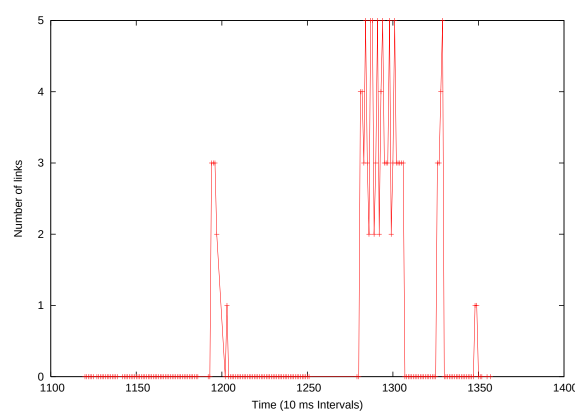

Figure 9. Number of links with utilization greater than 30 percent

as to stress efficiency issues. Because our 2D elements take slightly under a microsecond of CPU time per timestep, this is less than 5 seconds of serial work per timestep.

Figure 10 shows the predicted execution time per step, simulating 125 to 32,000 target processors using only 32 Lemieux processors. The simulated network is Blue Gene/L style three dimensional torus network. The simulation assumes latency-based network model without network contention. The time per step is 17.48 milliseconds for 125 processors and drops to 185 microseconds on 32,000 processors. Figure 11 is the corresponding speedup, normalized based on the 125 processor time. It shows that the program can scale well to at least several thousands of processors.

Beyond several thousand processors, when the simulated time per step drops below one millisecond, the parallel efficiency begins to drop. Sub-millisecond cycle times are indeed extremely challenging even on today's small machines, and we continue to seek methods to improve this performance on even larger machines.

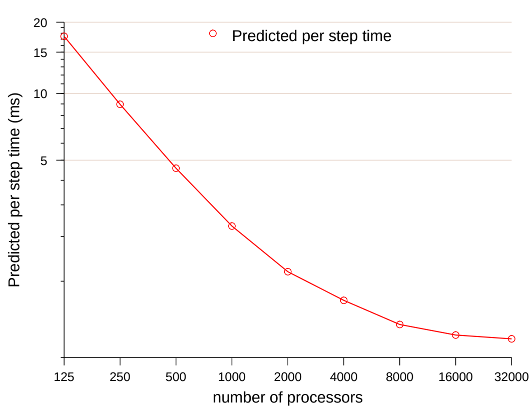

Figure 10. Predicted execution time

We also demonstrate the benefits of processor virtualization in CHARM++ for the same FEM program. We use different numbers of MPI virtual processors, each with a separate chunk of the problem mesh, on each simulated processor. Larger number of MPI virtual processors with finer decomposition on a simulated processor results in higher degrees of virtualization. Virtualization allows dynamic over-

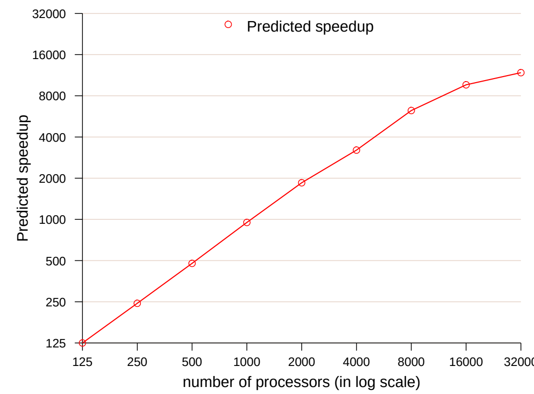

Figure 11. Predicted speedup

lap of computation and communication, and can improve cache utilization because each virtual processor's dataset is small.

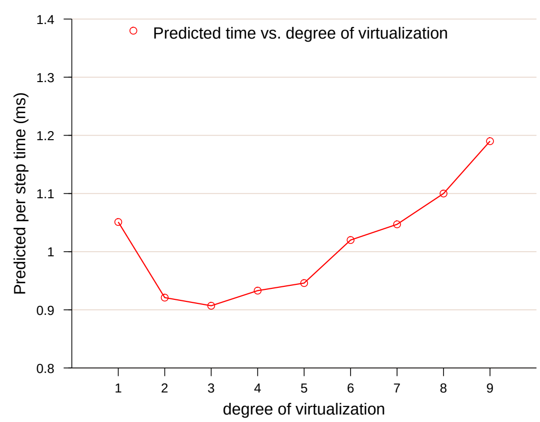

Figure 12. Predicted execution time vs. degree of virtualization

The predicted performance for various degrees of virtualization is illustrated in Figure 12. The problem size in this test is still the same—a 5 million element mesh, and the simulated machine size is fixed at 2000. Even a low degree of virtualization dramatically improves performance by allowing computation and communication to be overlapped; higher degrees of virtualization provide little benefit, and eventually the overhead of additional virtual processors only slows the program down.

5 Performance of Post-mortem Simulator

To evaluate the parallel performance of the simulator itself, we used the BigSim emulator on 16 real processors to run a 2D Jacobi program on 8000 simulated processors. This emulation generated log files that we then simulated by the POSE postmortem simulator running the simple latency-based network model using a varying number of processors. We show a speedup plot for the POSE simulator from 1 to 64 processors in Figure 13. The simulator had an average grainsize of 200 microseconds. The figure shows a significant improvement in simulator speedup over what we previously reported [15]. The best prior speedup relative to sequential time was just over 8 on 64 processors. Now we have a speedup of nearly 16 on 64 processors. Much of this improvement due to reductions in overhead achieved by POSE which results in better times on fewer processors. Thus speedup of the simulation relative to the one-processor parallel simulation versus speedup relative to sequential time are now very close. The one-processor parallel time for this run was only slightly slower than the sequential time.

Simulation of the more detailed contention-based network models is still quite challenging, but we are achieving decent speedups relative to one-processor parallel time as shown in Figure 14.

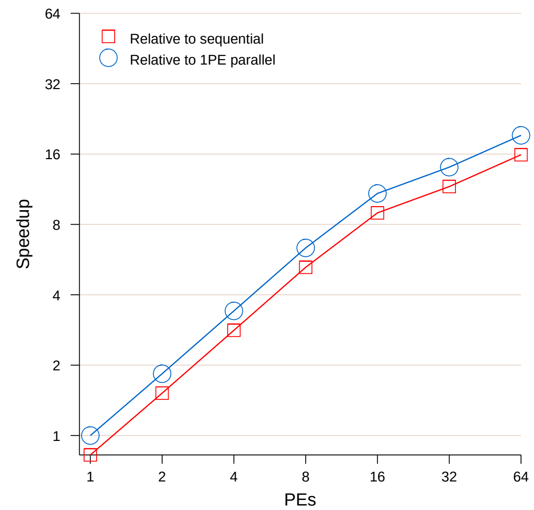

Figure 13. Speedup of Postmortem Simulation of Jacobi with Simple Latency-based Network Model

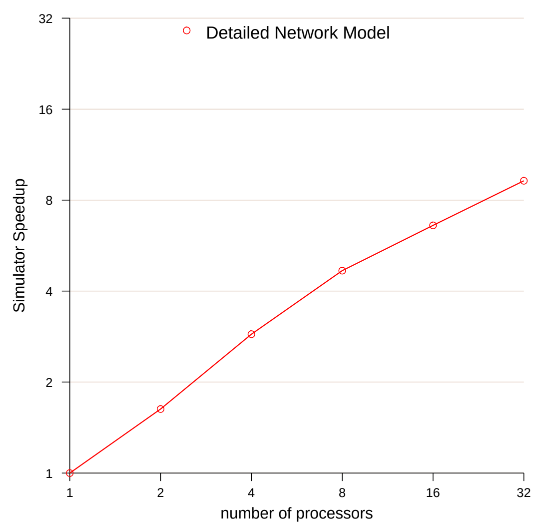

Figure 14. Speedup of Postmortem Simulation of NAMD with Detailed Contention-based Network Model

6 Conclusion and Future Work

It is clear that novel parallel programming models will be required to program petaflops class machines. This paper, along with the work in previously published papers, presents a programming environment for petaflops machines and Blue Gene. The programming environment is powered by the idea of processor virtualization in Charm++'s parallel migratable objects and Adaptive MPI. Issues faced while porting and developing such environments to such large machines can be dealt with ahead of the machine's availability by using a full scale emulator that we developed, using a recursive application of processor virtualization idea. The performance of parallel applications written for future petaflops computers can be predicted using the BigSim simulator either in coarse-grained or fine-grained mode with network simulation that models contention. For configuration without a co-processors, the behavior of messaging layers needs to be modeled in detail. The parallel applications that have been developed and evaluated in this environment include Molecular Dynamics simulation and Finite Element Method simulation. Future work will focus on increasing simulation accuracy and improving the scalability of the parallel applications, using compiler support to simplify the programming process further, as well as testing and refining the programming environment on the next large parallel machine, Blue Gene/L.

References

[1] M. Bhandarkar and L. V. Kalé. A Parallel Framework for Explicit FEM. In M. Valero, V. K. Prasanna, and S. Vajpeyam, editors, *Proceedings of the International Conference on High Performance Computing (HiPC 2000), Lecture Notes in Computer Science*, volume 1970, pages 385–395. Springer Verlag, December 2000.

[2] C. Ding and Y. He. A ghost cell expansion method for reducing communications in solving pde problems. In *SuperComputing 2001 Technical Papers*, 2001.

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

[4] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto. Time warp operating system. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 77–93. ACM Press, 1987.

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

[6] L. Kalé, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151:283–312, 1999.

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

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

[9] M.Blumrich, D.Chen, P.Coteus, A.Gara, M.Giampapa, P.Heidelberger, S.Singh, B.Steinmacher-Burow, T.Takken, and P.Vranas. Design and analysis of the bluegene/l torus interconnection network. Technical report, IBM Research Division, T.J.Watson Research Center,P.O.Box 218, Yorktown Heights, NY 10598, 2003.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] G. Zheng, T. Wilmarth, O. S. Lawlor, L. V. Kalé, S. Adve, D. Padua, and P. Guebelle. Performance modeling and programming environments for petaflops computers and the blue gene machine. In *NSF Next Generation Systems Pro-*

gram Workshop, 18th International Parallel and Distributed Processing Symposium(IPDPS), Santa Fe, New Mexico, April 2004. IEEE Press.