Performance Modeling and Programming Environments for Petaflops Computers and the Blue Gene Machine

Gengbin Zheng, Terry Wilmarth, Orion Sky Lawlor, Laxmikant V. Kalé, Sarita Adve, David Padua
Dept. of Computer Science
University of Illinois at Urbana-Champaign
{gzheng, wilmarth, olawlor, kale, sadve, padua}@cs.uiuc.edu

Philippe Guebelle
Dept. of Civil Engineering
University of Illinois at Urbana-Champaign
geubelle@uiuc.edu

Abstract

We present a performance modeling and programming environment for petaflops computers and 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, an ongoing effort to incorporate a pluggable module of a detailed contention-based network model. It provides the ability to make performance predictions for machines such as BlueGene/L. We also explore the programming environments for several planned applications on the machines including Finite Element Method (FEM) simulation.

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 128,000 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. Further, it is also important to understand performance issues in specific algorithms thoroughly, so next-generation applications can be built to scale to such large machines. We have been engaged in a project to address these challenges for over two years. In this paper we summarize our progress and findings so far, with focus on recent unpublished results.

We explored the CHARM++ programming model as an appropriate model for large machines because of its ability to virtualize processors [1], 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. Further, CHARM++ provides a solution to the issues that arise due to fine-grained computations resulting from using large machines. We first describe issues, explored using an emulator, in scaling CHARM++ and Adaptive MPI [2] (built using CHARM++) to run on large machines. Next we present our performance prediction system, based on parallel discrete event simulation, and novel ideas to avoid re-execution during optimistic simulation. Recent performance results using the simulator for structural dynamics computations involving the Finite Element Method (and unstructured grids) are discussed next. Progress on detailed architecture simulation of multi-processor nodes, which is needed to accurately predict individual processor performance in the context of a large simulation is then summarized, followed by an overview of future and ongoing research issues in the final section.

2 Programming Petaflops Machines

Deciding the characteristics of an ideal programming environment for a massively parallel machine like Blue Gene is a challenging task. This is because dealing with tens of thousands or even millions of processors requires qualitative change in both the programming environment and the runtime system.

In this context, we have developed a multi-tier programming model in which the object layer forms a middle layer which is supported from below by a low level explicit model, and which supports higher level components including domain specific languages and libraries. This section briefly describes the middle and lower layers. The higher level is presented in section 4.1.

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. A runtime library call allows a thread to send a short message to a destination node. The header of each message encodes a handle function to be invoked at the destination. A designated number of threads continuously monitor the incoming buffer for arriving messages, extract them and invoke the designated handler function. We believe this low level abstraction of the petaflops architectures is general enough to encompass a wide variety of parallel machines with different numbers of processors and co-processors on each node.

We have developed a software emulator based on this low level model. The details of the emulator and its API were presented in [3].

In this base level model, the programmer must decide which computations to run on which node. The programming environment at a higher level relieves the application programmer of the burden of deciding where the subcomputations run.

In this context, we have evaluated the CHARM++ as a parallel programming language for petaflops machines and also as an alternative to 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[4], 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 in [5].

Adaptive MPI, or AMPI, is an MPI implementation and extension based on CHARM++ message driven system, that supports processor virtualization[1]. AMPI implements virtual MPI processes (VPs), several of which may be mapped to a single physical processor. Taking advantage of CHARM++'s migratable objects, AMPI also supports adaptive load balancing by migrating MPI threads.

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

3 Performance Modeling

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.

We have developed BigSim[6] for simulating petaflops class machines such as Blue Gene/L. In the rest of this section, we will first present the BigSim simulator, followed by BigNetSim. BigNetSim is a work in progress that extends BigSim with network simulation capability.

3.1 BigSim

Performance of parallel applications is known to be difficult to predict due to the 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, causing causality errors and destroying the accuracy of the simulation.

Traditional methods of correcting this involve high synchronization overheads as are often found in optimistic concurrency control. These overheads include: (a) checkpointing overhead, (b) rollback overhead and (c) forward execution overhead. In BigSim, taking advantage of the parallel program's inherent determinacy, we are able to greatly improve the simulation efficiency by dramatically reducing the overheads. The details of BigSim are described in [6].

BigSim is based on direct execution of an application on the emulator described previously. Our approach for the simulation involves letting the emulated execution of the program proceed as usual, while concurrently running a parallel algorithm that corrects time-stamps of individual messages.

BigSim is a coarse simulator in that the network contention is ignored although path-dependent message latencies are modeled. This is often found adequate for major class of computation bounded applications. However, for applications that require detailed network modeling, the simulator needs to be extended to model network contention. Thus we present an extension to BigSim, the BigNetSim simulator, an ongoing effort to incorporate detailed network simulation in performance prediction.

We next describe the PDES engine POSE used in the development of BigNetSim and our progress so far.

3.2 Postmortem Simulation

BigNetSim takes a different approach by performing postmortem simulation. The BigSim emulator generates a log of tasks that were performed and their dependencies. However, the tasks in this log have not been ordered in time with respect to each other, and need to be timestamp-corrected in order to obtain performance results about the original program that was run on the BigSim emulator. To perform timestamp correction, we use a parallel discrete event simulation (PDES) environment called POSE[7][8].

BigNetSim plugs into the timestamp correction simulation and simulates the same application over a detailed network model. The behavior of the network model can be varied by its input parameters to model a variety of situations without ever needing to re-run the original simulation.

3.2.1 POSE

Our PDES environment is built in CHARM++. CHARM++ supports the *virtualization* programming model, an approach we believe will give rise to great improvements in PDES performance. Virtualization involves dividing a problem into a large number N of components that will execute on P processors[1]. N is independent of

P, though ideally $N >> P$. The user's view of the program consists of these components and their interactions; the user need not be concerned with how the components map to processors. The underlying run-time system takes care of this and any subsequent remapping (see Figure 1).

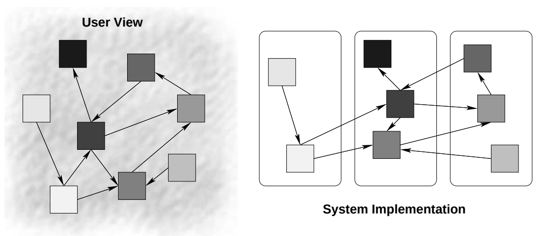

Figure 1. Virtualization in CHARM++

In CHARM++, these components are known as *chares*. Chares are C++ objects with methods that may be invoked asynchronously from other chares. Since many chares can be mapped to a single processor, CHARM++ uses *message-driven execution* to determine which chare executes at a given time. This is accomplished by having a dynamic scheduler on each processor. The scheduler has a pool of *messages*, i.e. method invocations destined for a particular chare, and selects one of these, determines the object it is destined for, and then invokes the corresponding method on the object. At completion, the scheduler selects the next message. Different scheduling policies are available, as well as *prioritized execution*, which enables the user to attach priorities to messages. The advantage of this approach is that no chare can hold a processor idle while it is waiting for a message. Since $N > P$, there may be other chares that can do work in the interim.

The logical processes (LPs) of PDES can be mapped onto CHARM++'s chares in a straightforward manner. Similarly, we use timestamps on messages as priorities and thus the CHARM++ scheduler becomes 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.

Posers 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.

Posers can pass simulated time in two ways. First is the *elapse* function. This is used to pass a certain amount of local time (presumably performing some activity). It advances the OVT of the poser in which it is called. The second means is by an *offset* added to event invocations. This can be used to schedule a future activity or to indicate *transit time* in a simulation. For example, suppose the event being

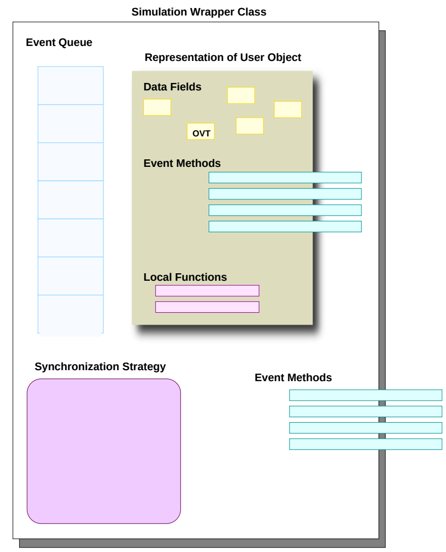

Figure 2. Components of a poser.

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.

Posers have *plug-in behaviors* for their underlying implementation. For the user, it is enough to know that the implementation involves the queuing of events, the synchronization of their execution, and access to and modification of poser state. We refer to these respectively as the **wrapper** behavior, the **synchronization strategy** behavior, and the **representation** behavior. Different approaches can be used for each. Figure 2 illustrates how these components fit together. The simulation developer concentrates on modeling entities (the representation). For more control over simulation behavior and performance, the developer can try different synchronization strategies.

Why encapsulate so much information in each entity, and then encourage the breakdown of the physical system into many such entities? Virtualization allows us to maximize the degree of parallelism. 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.

Speculative Synchronization POSE makes use of optimistic concurrency control which originated as Time-Warp[9]. 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*[10] 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, the behavior is similar with some exceptions. First, 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.

In POSE, events are inserted into the event queue on the object for which they are destined. When the object gets control, it invokes the synchronization strategy to process events. Typically, the event just received will be the event processed, and no other events will exist on that object. However, some events may arrive slightly ahead of schedule and may be worthy candidates for speculative execution. Others may arrive with timestamps very far in the future and it may be unwise to execute them speculatively.

The synchronization strategy determines how to process the events within a speculative window. To avoid rescheduling overhead, we allow objects freedom to speculate once they get control. Event processing on the object follows this pattern: if there are events with timestamp $t \geq$GVT but within the speculative window, do *all* of them. The later events are probably not the earliest in the simulation, and it is very 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 is a work in progress. We are developing an adaptive strategy that strives to execute more events at a time while minimizing rollbacks. This strategy will detect object behavior patterns and adjust to them by altering speculative window size and event processing behavior.

3.2.2 Timestamp Correction

For the timestamp correction simulation, we read the trace log files generated by the BigSim emulator. An application execution was emulated on some configuration, and all the tasks and their dependencies 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 latency which we use to determine how much time generated tasks spend in transit to the processor on which they will be executed.

What we don't 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 application that was emulated 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.

```
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.

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 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 GVT should represent a correct runtime for the emulated application.

3.2.3 Big Network Simulation

BigNetSim takes the POSE timestamp correction simulation to the next level. Instead of using some preset latency value to determine message transit time, we actually model the message as it passes through a detailed contention-based network model. The power of this approach is that we could model any type of network we wish and plug it into the original timestamp correction simulation and get new results. This enables us to run the application emulation once, and reuse the trace logs generated by the emulation to repeatedly analyze the application in a variety of network configurations. Figure 4 shows how these components interact.

For our network simulation, the nodes are configured with dimensions as originally given to the emulator. Each node has a switch which routes message packets through the network via six channels. Communication is achieved by first trying to lock a path of channels between switches from the source to the destination. Once a path is obtained, the switch sends out the packets on the appropriate outgo-

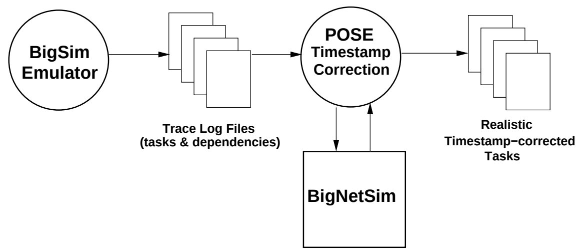

Figure 4. Interaction between BigSim, POSE timestamp correction and BigNetSim

ing channel. After the last packet is transmitted, the path is unlocked one channel at a time, following the last packet.

BigNetSim is currently parameterized by bandwidth, wraparound cost (for channels from edge and corner nodes to other edge/corner nodes), and maximum packetsize.

4 Parallel Libraries and Domain Specific Frameworks

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 planned applications for Blue Gene/L. It is a challenging application to parallelize. A microsecond simulation includes about a billion timesteps. Thus, each timestep involves a relatively small amount of computation that must be effectively parallelized.

We have developed an experimental prototype program called LeanMD that models the essential computations of MD. Performance study and analysis of LeanMD on BigSim for Blue Gene/L in [6] shows that LeanMD can scale up to petaflops machines.

In this section, we focus on the parallel FEM framework and its simulation on massive parallel machines.

4.1 Big FEM Simulation

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[11], 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.

There are several obstacles to solving FEM problems on very large machines. First, we must generate a mesh of sufficient size; second, unless the mesh is enormous, we must deal with the problem's small grainsize.

4.1.1 FEM Meshes for Large Machines

Consider a high-resolution fracture dynamics simulation of a block of metal. We first decompose the block of metal into "elements", which are small pieces of the domain with a simple shape, often tetrahedra or cubes. In theory, the number of elements is determined only by the physical fidelity we wish to achieve, but in practice the number of elements, and hence the accuracy of the simulation, is often limited by the memory and speed of the machine, as summarized in Table 1.

For example, a 1-meter cube of metal discretized into a 1cm scale mesh will require one million elements. But 1cm is quite coarse; if we need 1mm resolution the mesh will have one billion elements. If elements require 40 bytes each, such a mesh would require 40 GB of storage. This is larger than current serial machines can handle, but is plausible even on today's parallel machines.

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 currently uses the serial Metis partitioning library, so the partitioning is performed completely on one processor, which becomes a bottleneck for large meshes. We are working on integrating the parallel ParMetis partitioning package to avoid the serial mesh partitioning bottleneck, which should allow 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.

4.1.2 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. As can be seen in Table 1, 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 each message takes on the order of 10us, processors will spend all

Resolution	*Elements*	*Storage*	*1 Processor*	*1K Processors*	*100K Processors*	*1M Processors*
1cm	10^6	40MB	1s	1ms	10us	1us
1mm	10^9	40GB	1000s	1s	10ms	1ms

Table 1. Meshes for a 1m^3 domain, storage requirements and time per timestep on various machines.

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 [12], 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.

4.1.3 Bottlenecks on Large Machines

In summary, there are a number of practical bottlenecks to execution on very large machines. First, large meshes must be generated; this is difficult with today's tools. Second, the meshes must be partitioned for parallel execution. Finally, the resulting computation may still have small grainsize, so messaging performance is important.

Our runs with BigSim also exposed a number of unexpected bottlenecks and limitations to scalability. For example, the serial partitioning library we use 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. Hopefully ParMetis will solve this problem.

Similarly, even though our MPI implementation, AMPI, was designed to be scalable, while trying to simulate very large machines we discovered our implementation used P^2 total memory for P processors. The culprit was a simple linear message ordering table kept by each processor, because the table's length was proportional to the number of processors. For today's machines, where P=1000, the total amount of memory used was 16MB; but for P=100,000, the tables would use 160GB! The solution was to break the tables into (software) pages and only allocate pages when referenced; this dramatically reduces the storage requirements for large machines because most processors only communicate directly with a small subset of other processors.

5 Performance

We first present results of validation tests using BigNetSim on Lemieux [13] at Pittsburgh Supercomputer Center. We then present results of performance prediction and performance analysis of some real world FEM applications using the simulator. Finally we will present the scaling performance of the BigNetSim simulator itself.

5.1 Validation

We have compared the actual running time of a simple 7-point stencil computation with a 3-D decomposition written in MPI with our simulation of it using BigNetSim. In the program, every chunk of data communicates with its six neighbors in three dimensions. The Jacobi relaxation computation is performed, and the maximum error is calculated via **MPI_Allreduce**.

The result is shown in Table 2 for a problem with fixed size in all runs. The first row shows the running time of the MPI program on 32 to 256 processors; the second row shows the predicted running time using BigNetSim offline 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	32	64	128	256
Actual run time (s)	2.21	1.07	0.48	0.26
Predicted time (s)	2.35	1.16	0.55	0.30

Table 2. Actual vs. predicted time

5.2 FEM

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

Figure 5 shows the predicted execution time per step, simulating 125 to 16,000 processors using only 32 Lemieux processors. The time per step is 23.5 milliseconds for 125 processors and drops to 640 microseconds on 16,000 processors. Figure 6 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 a few milliseconds, 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.

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. Virtualization allows dynamic overlap of computa-

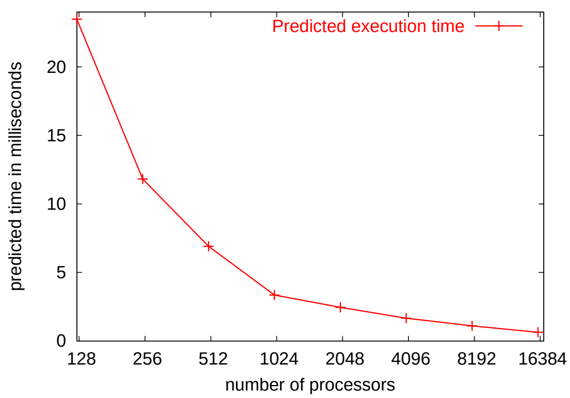

Figure 5. Predicted execution time

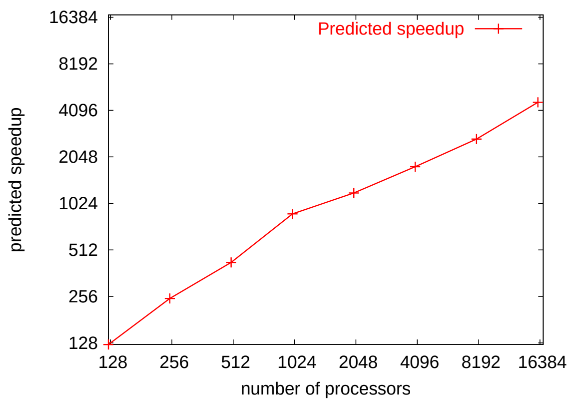

Figure 6. Predicted speedup

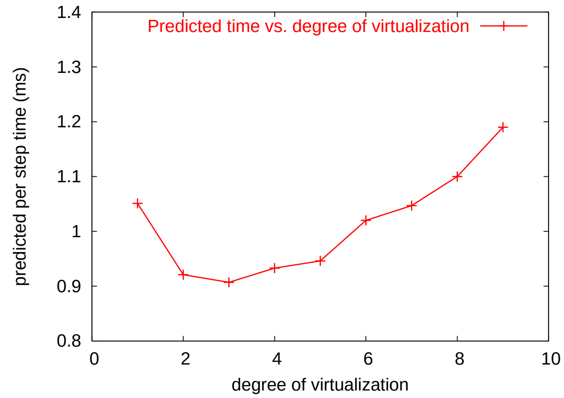

Figure 7. Predicted execution time vs. degree of virtualization

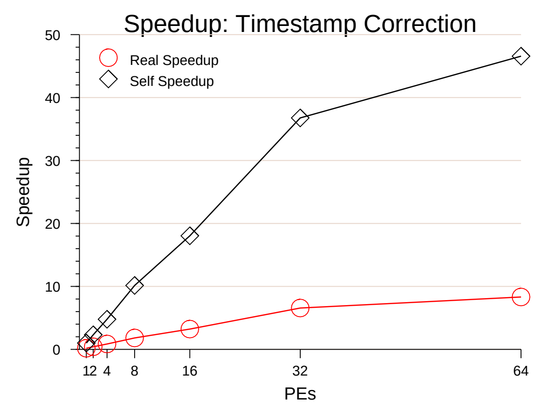

Figure 8. POSE Timestamp Correction Simulator Speedup

tion and communication, and can improve cache utilization because each virtual processor's data is small.

The predicted performance for various degrees of virtualization is illustrated in Figure 7. The problem size in this test is still the same—a 5 million element mesh, and the simulated machine size is fixed at 2000 [1]. 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.

5.3 Performance of Post-mortem Simulator

To evaluate the parallel performance of the simulator itself, we used the BigSim emulator on 32 real processors to run a 2D Jacobi program on 8000 simulated processors. This emulation generated trace log files that we then loaded into the POSE timestamp correction simulator. We show a speedup plot for the POSE simulator from 1 to 64 processors in Figure 8. The simulator processed 5,085,836 events and had an average grainsize of 198 microseconds.

The figure shows two plots: *real* speedup and *self* speedup. Self speedup shows how the program speeds up with respect to itself; i.e. the single processor time is the time for the parallel POSE simulation. Real speedup uses an ideal sequential time estimate of how long it would take to execute just the events of the simulation with no overhead for timestamp sorting; i.e. the time the program would take if we knew in advance the exact order in which to execute the events, as well as which events are to be executed. This is a lower-bound on the sequential time for the simulation. As the figure shows, self speedup is nearly perfectly linear up to 32 processors and tapers off after that, while real speedup shows a modest but correspondingly steady speedup improvement as we add processors.

6 Architectural Simulation

To further improve the accuracy of performance prediction, it is necessary to accurately predict the timings of sequential fragments of codes, with instruction level accuracy. We are exploring the idea of incorporating an architecture simulation of multi-processor nodes via RSIM [14] simulation infrastructure, and its enhancements.

In the context of such large-scale simulations, it is quite challenging to exploit such a hardware simulator in BigSim.

[1]Our current partitioning scheme limits the number of partitions. So, the machine has to be small to allow a high degree of virtualization

This is because detailed microarchitecture-level simulation runs an order of magnitude slower than other multiprocessor simulations that do not model the processor in detail.

To overcome the speed penalty, we have enhanced RSIM with a fast functional simulator called Rabbit. RSIM is used in Rabbit mode to accelerate initialization and other portions of the code where it is not important to collect timing statistics. The performance results show that the acceleration provided by "Rabbit Mode" is quite significant[14].

The second enhancement to RSIM models on-chip multithreading which is likely to be a key feature of future high-performance chips. Capabilities of simulating chip level multiprocessors has also been added to RSIM. We have an initial version of RSIM that supports detection of application phases [15], allowing it to predict performance of some phases without having to simulate them in detail.

With these enhancements to RSIM, we are integrating RSIM into BigSim to improve the simulation accuracy to the instruction level.

7 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. The performance of parallel applications written for future petaflops computers can be predicted using the BigSim simulator either in coarse grain or fine grain mode with contention-based network simulation. 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.

References

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

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

[3] 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.

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

[5] Gengbin Zheng, Arun Kumar Singla, Joshua Mostkoff Unger, and Laxmikant V. Kalé. A parallel-object programming model for petaflops machines and blue gene/cyclops. In *Next Generation Systems Program Workshop, 16th International Parallel and Distributed Processing Symposium(IPDPS), 2002*, Fort Lauderdale, FL, April 2002.

[6] Gengbin Zheng, Gunavardhan Kakulapati, and Laxmikant V. Kalé. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *2004 IPDPS Conference, 18th International Parallel and Distributed Processing Symposium*, Santa Fe, New Mexico, April 2004.

[7] Terry Wilmarth. Pose: A study in scalable parallel discrete event simulation. Technical Report 03-16, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, May 2003.

[8] Terry Wilmarth and L. V. Kalé. Pose: Getting over grainsize in parallel discrete event simulation. Technical Report 04-01, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, January 2004.

[9] 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.

[10] L.V. 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.

[11] Milind 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.

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

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

[14] Christopher J. Hughes, Vijay S. Pai, Parthasarathy Ranaganathan, and Sarita V. Adve. Rsim: Simulating shared memory multiprocessors with ilp processors. *IEEE Computer, special issue on simulation*, pages 40–49, February 2002.

[15] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 30th International Symposium on Computer Architecture ISCA'2003*, 2003.