

Using SPEEDES to Simulate the Blue Gene Interconnect Network

Introduction

In November 2001, IBM announced a jointly funded research partnership between IBM and the Lawrence Livermore National Laboratory as part of the United States Department of Energy ASCI Advanced Architecture Research Program. This partnership targets the production of a massively parallel machine, BlueGene/L (BG/L), scheduled to be operational in the 2004-2005 time frame. BG/L is a scalable system in which the maximum number of compute nodes assigned to a single parallel application job is $2^{16} = 65,536$. The BG/L nodes exploit system-on-a-chip technology, configured in a $64 \times 32 \times 32$ 3-D torus, to deliver target peak processing power of 360 teraFLOPS (trillion floating-point operations per second) at price/performance and power consumption/performance targets unobtainable with conventional architectures.

Caltech's Jet Propulsion Laboratory (JPL) and Center for Advanced Computer Architecture (CACR) is conducting application and simulation analyses of BG/L in order to establish a range of effectiveness for the Blue Gene/L MPP architecture in performing important classes of computations and to determine the design sensitivity of the global interconnect network in support of real world ASCI application execution.

Approach

A number of applications are being analyzed for the purpose of providing workload traces for a statistical BG/L simulator. The selected applications stress load balancing, multiple languages, and dynamic behavior with respect to CPU/memory/communications usage throughout execution. In particular, the applications and kernels being traced and analyzed are:

- Magnetic Hydro Dynamics (MHD) - A fluid solver for equations of hydrodynamics and resistive Maxwell's equations. This application decomposes physical domains into regular domains across processors, exercising nearest neighbor communications and global reductions each time step.
- Quantum Monte Carlo (QMC) – The code calculates material properties within chemical accuracy. A more efficient implementation of this application is under development, yielding huge computational demands, small memory and communication requirements.
- Gyrokinetic Toroidal Code (GTC) – GTC calculates micro-turbulence in a tokamak using kinetic equations in a collisionless regime. The code is highly compute bound, except for synchronizing MPI Allreduce calls.
- 3-D Adaptive Mesh Refinement (AMR3D) – A fluid dynamics code performing Richtmyer-Meshkov shock simulation using GrACE data management libraries

for mesh generation and automatic load balancing. AMR3D is highly dynamic and generates collective and point-to-point message traffic in bursts, during re-gridding events.

- 2-D AMR MHD – An elliptic iterative solver for ideal MHD equations, using AMR finite difference calculations. This code allows compelling comparisons between large unimesh (no AMR) traces and AMR (increasing levels of refinement) enabled runs. Figure 2 shows communication traffic traces during execution of an 128x128 mesh, with 2 levels of AMR run on 32 processors. The communications workload was captured by CACR's Expandable Trace Format (ETF) tool, enabled by simple application instrumentation.

The direct approach of cycle-by-cycle level simulation a 64K node system running parallel applications is infeasible due to limitations of existing computer systems. In lieu of this, the JPL/CACR team is taking a statistical approach using parameterized models of the applications (workloads) and statistical (queuing) models of processing node message traffic derived from traces produced by the computational experiments. All 64K nodes will be explicitly represented, but message traffic will be aggregated to reduce simulation run time while retaining statistical effects of adaptive routing and network contention as a function of network load and size.

Commercial sequential discrete event simulation packages are incapable of handling systems of 64K nodes. For example, sequential discrete event statistical model was developed using a commercial modeling tool (SES/workbench). Due to memory limitations, this sequential model is scalable up to only 1024 BG/L nodes. To cover the full range of BG/L sizes it was necessary to exploit parallel discrete event simulation software that could then be run on parallel machines for scalability.

Parallel discrete event simulation has been successfully used for battlefield simulations which may involve large numbers of complex simulation objects (millions) where processing of individual events is compute intensive. Parallel simulations of MPP architectures and computer networks in which a large number simple objects and small compute time events are processed have not been as widely studied. A parallel discrete event statistical model was developed to scale to the full 64K nodes. Both optimistic and time step methods are being used to speed up the parallel simulation. An optimistic time step model was developed using the SPEEDES (Synchronous Parallel Environment for Emulation and Discrete-Event Simulation) framework developed at JPL. Results of this approach are reported in this paper.

Background

Discrete event simulation is a simulation model in which different simulation objects exist. These objects are executed at discrete points in time. Objects pass simulation time-stamped event messages to other objects (or to themselves) that cause the recipient objects to execute at the point in simulation time carried by the message. When the receipt of a message causes an object to execute at a point in simulation time, this action taken is called an event. Discrete event simulation differs from a time-step simulation in

which objects are checked at each time step in order to assess and modify the current state of the object

It is often possible to develop a simulation as either a discrete event simulation or a time-step simulation. For example, a colliding pucks type of simulation could be developed as a time-step simulation by calculating the position of each puck at each time step, based on its state in the previous time step. Alternatively, in a discrete event based simulation of the same application, collision events could be scheduled based on a puck's current position relative to the boundaries of the table as well as the positions of other pucks. In the latter case, each puck's state is only updated when a specific event occurs, not at regular time steps. When the system being simulated is by its nature asynchronous, it is usually best to use the discrete event model. Forcing the use of a time-step model in that case may result in inefficiencies, because of the overhead that occurs as each time step is executed whether or not any work needs to be done in that time step.

In a discrete event simulation, care must be taken that all events are executed in the proper time order. This is typically accomplished in a sequential implementation by forming a sorted event list, from which events are taken one at a time to be executed. The situation in a parallel discrete event simulation (PDES) is harder. While individual event lists on each node are ordered in time, something must be done to in effect order those lists across all the nodes involved in the simulation. Consider two events, E_1 and E_2 , where E_1 occurs at an earlier simulation time than E_2 , and the outcome of E_1 affects the execution of E_2 . If the two events are executed on separate nodes, something has to be done to coordinate the execution of both. If E_2 is allowed to run first, without having the information that E_1 is to provide, a *causality* error may result.

Mechanisms in parallel discrete event simulation to deal with possible causality errors generally fall into two categories: conservative and optimistic[1]. Conservative algorithms avoid causality errors by constraining the operation of the simulation so that (in the above case) E_2 is not allowed to execute until E_1 has completed. Such constraints can come at the cost of efficiency, so that less than optimal parallelism may be extracted from the simulation.

Optimistic algorithms, on the other hand, fully exploit available parallelism by allowing causality errors to occur, but the algorithm will detect this situation and force event rollbacks as necessary. In the previous example, if E_2 executes before E_1 , when the simulation framework executes E_1 it detects that E_2 must be rolled back to the state it was in before E_1 finished execution. The optimistic algorithm then re-executes E_2 , this time including any state changes caused by E_1 . This does of course lead to additional overhead, and a requirement for a more complicated simulation framework that can handle rollbacks, but practice has shown that this cost is generally outweighed by the extra parallelism extracted.

SPEEDES is an optimistic simulation framework developed at the Jet Propulsion Laboratory by Dr. Jeffrey Steinman in the early 1990's. It has been put to use by DOD in numerous battlefield simulations. By default, SPEEDES uses a synchronization algorithm called *breathing time warp* based on the concept of virtual time developed by

Dave Jefferson[2]. SPEEDES modifies Jefferson's original Time Warp concept by placing a limitation on the number of rollbacks that may occur in the course of the simulation. This algorithm uses a time window to prevent runaway objects from generating excessive numbers of rollbacks. However the choice of algorithm is governed by a runtime parameter that may be modified to remove any such limitations, allowing a pure Time Warp based algorithm to be used.

When it came time to install SPEEDES at JPL, we were first faced with the decision of whether to install it on an SGI Origin 2000, or a Beowulf style computer. The program's author recommended the Origin because it uses a shared memory model for communications between processors, which, for SPEEDES, is much more efficient than using a message passing model such as MPI. In addition, we were told that a SPEEDES simulation had previously been developed that handled 1,000,000 simulation objects running on 100 Origin nodes. This told us that SPEEDES would scale efficiently, and that the choice of the Origin would work well for our purposes. Because SPEEDES had previously been ported to the Origin, it was fairly straightforward to install the source code on our computer and compile and test it.

Model Development

IBM developers had previously written a simulation of the Blue Gene interconnect network in order to determine basic design parameters and functionality. That simulation was written using C, without the benefit of a simulation framework. It was written as a parallel discrete event simulation, and used a conservative time management algorithm. The approach was made efficient by grouping events into time buckets, and synchronizing nodes after processing all events in the bucket. Speedup on 16 nodes was close to linear when the workload was a random one, because of the fairly even distribution of work across the nodes. For a workload generating an uneven distribution of messages, however, speedup dropped to 5.6.

One of the goals of the IBM simulator was to simulate all network activity at a cycle-by-cycle level of granularity. In this it was very thorough, but because of its high fidelity large workloads could not be handled in a reasonable amount of time. This motivated the development of a simulation model that would use a coarser level of granularity, and would be able to handle larger workloads. Ultimately we would like to simulate millions of packets being transferred between 65,536 simulated nodes, in order to understand the response of the network under a variety of loads.

We expect that a PDES using optimistic time management will be more efficient than conservative methods at handling workloads involving uneven workload distributions, especially for the case where the load imbalances may exist at any instant in simulation time, but that over the course of the entire simulation the total loads on each node balance. This is because optimistic methods allow for *temporal load balancing*. Nodes are allowed to run at different rates, so that if a node temporarily has a heavier load, it is allowed to fall back in simulation time, and can catch up later when its load is eased.

In part to begin to familiarize ourselves with the way SPEEDES operated, and in part to determine whether it would scale properly for our application, we started with a very simple model. Once it was working correctly and efficiently, we built additional features into it. This approach helped to highlight the effect any newly added features had on the efficiency of the simulation, and facilitated the debugging process.

Because of the complexities involved in optimistic simulations, a good simulation framework needs to handle these in a way that is as transparent as possible to the user. SPEEDES does all the work of detecting causality errors, and handles rollbacks transparently.

The fundamental programming construct in SPEEDES is the simulation object. Each such object communicates to other objects by sending and receiving time-stamped messages. Receipt of a message eventually triggers the receiving object to process that message. The simulation time of the received message becomes the simulation time associated with the corresponding object when it executes. If an object executes during a certain time slice and is not rolled back, that event is said to be *committed*.

In our simulation we model each network node as a unique SPEEDES object. Each network packet is modeled as a message in the simulation. We decided to design at this level of granularity because of the complexity involved in the adaptive routing being used by the network. As congestion builds up in one part of the network, traffic patterns change in an attempt to route messages around the congested area. This behavior is difficult to encapsulate analytically, and we felt that the model itself should behave in this manner.

Our simulation messages are quite small, and only contain information relating to the transfer process, such as the origination node, destination node, time of origination, and so on. When a message is sent from one object to another, it triggers an arbitration process in the receiving object that determines whether the message needs to be sent on, and if so, what route it should take. In parallel with this, the same object is examining its workload queue to determine whether new messages are being generated. Information about congestion on that node is sent back to the originating node to assist in flow control and the adaptive routing algorithm.

SPEEDES is written in C++, and uses the object-oriented nature of the language to good effect in hiding some of the complexity involved in state-saving. Through the use of macros, it provides “rollbackable” equivalents of most common variable types. By declaring a variable to be of one of these types, the user is relieved of having to be concerned about saving and restoring the state of that variable. For example, the common C++ types of `int` and `double` have equivalents named `RB_int` and `RB_double`. Through operator overloading provided by SPEEDES, they can be used in exactly the same way as their counterparts, but in addition will have their values saved automatically, and restored whenever the object undergoes rollback. Other such types include `RB_logical`, `RB_string`, and `RB_ptr`. Memory allocation and deallocation routines that can be rolled back are also available. An inconsistency could arise if memory was allocated by an object, and an address to that memory were to be saved in a pointer. If

the object is rolled back, the reallocated memory might be in a different location, and the restored pointer would now have the wrong value. For this purpose SPEEDES implements *smart pointers* that can keep track of where allocated memory is really located, and adjust themselves accordingly.

Various classes of rollbackable storage are provided by SPEEDES. One and two dimensional arrays that are either fixed in size or dynamically sized are supported, as are one dimensional polymorphic arrays. In addition to support for arrays, SPEEDES provides two rollbackable container classes: a keyed list class and a keyed tree class. Because of the flexibility in the way the classes are implemented, and in the number of methods defined for each, our simulation was able to use the keyed list class as a FIFO type. Key usage is not required, and operations such as PushTop and PopBot lent themselves perfectly for our purposes of defining queues to hold packets as they arrived, and while they were waiting to be sent out.

Provision is made for committing output to files or “standard out” or “standard error” through the RB_ostream class. It behaves just like the equivalent C++ ostream class, except that the output to the stream is only sent once the event is committed.

As touched upon earlier, in our model an event is triggered every time a message is received at a simulated node, whether it is to be passed on by that node or consumed by that node. This kind of event accounts for the largest number of events in the model. With the fine level of granularity we use, most of our elapsed time is spent in the overhead SPEEDES requires to handle the events, and not much time is spent within the event itself. Consequently, limiting the number of events can do much to speed the elapsed time of the simulation.

One way we have found to limit the number of simulation events is to use the SPEEDES feature of persistent processes. In our initial try, each loop through the arbitration process triggered a new event, but this resulted in slow run times. SPEEDES has a special WAIT command (implemented as a macro) that forces a process to give up time, depending on the state of a semaphore. By setting the semaphore every time a message is available for arbitration, we are able to limit the number of events required by the arbitration cycle.

Though overall SPEEDES does a good job of shielding the programmer from most of the complications involved in PDES, some complications in designing such a simulation did come through. For example, it was not obvious from the documentation whether messages were persistent or whether space for them had to be allocated in a rollbackable fashion. In general, the user must be aware of when rollbackable memory needs to be used. Normally if the contents of the memory are to be used only in the course of an event, that memory need not be rollbackable, but this rule can be complicated when one uses special classes such as the keyed list class. In our simulation, whenever we stored messages in the list, or retrieved them for resending, the contents had to be copied between normal memory and rollbackable storage.

Some of the problems mentioned, as well as our optimization work, required trial and error and added code for debugging. Our Totalview and gdb debuggers did not work

well with this system, so we had to use other means. SPEEDES comes with a number of sample programs that can be compiled and run, that demonstrate various features. These turned out to be very helpful for filling in some gaps in the documentation. SPEEDES can be configured to print out statistics at the end of each run, such as the number of events handled by each routine. This feature was especially useful for those regular simulations where the correct number of events was predictable.

SPEEDES runs in a deterministic fashion, so that the same path is followed each time an identical simulation is run. This is particularly important for debugging. It also can be run in a trace mode, where each event is logged to an output file. The most challenging type of problem to debug were the crashes that resulted typically when SPEEDES routines were called inappropriately. Output statements such as `printf` and `cout` helped illustrate the path the code was taking and could narrow down the possibilities of which call was causing the problem. The previously mentioned `RB_cout` calls were not helpful, because they were only printed if the event in question completed and was committed. If a call within the event triggered the crash, the event would not complete and the `RB_cout` call would never succeed.

Results

The first model on which we measured performance was a very simple one, designed just to familiarize ourselves with the performance, scaling, and use of SPEEDES. As such, it did not have any message flow control or adaptive routing built into it. Instead it sent messages as fast as the hardware and software would allow, and used a simple dimension ordered routing algorithm. At the beginning of the simulation, each simulated node generated 1,000 packets, each of which was destined for a randomly chosen node in the network. In turn, each generated message was injected into the receiving routine for the originating node.

The receiving routine handled each incoming packet identically, whether it had just been injected, or had been received from an adjacent node. In either case, the destination node information was retrieved from the packet content, and a determination was made as to whether the packet was at its final destination, or whether it had to be sent on to another node. In the event that it had to be sent on, the routine determined what the next step was in the packet's routing, and sent the packet on its way.

The SGI Origin 2000 on which we ran the simulation has 128 R12000 processors available, each running at 300 MHz. These are configured as 2 processors per node. With each node containing 1 GB of RAM, it has a total of 64 GB of RAM.

Each message injected in the simulation resulted in an event. An event was also generated each time the message was received, whether it was received on the originating node when it was first injected, or received by a destination or intermediate node. The network in our model is a three dimensional torus that allows packets to be sent in either direction for each dimension. If x , y , and z represent the size of the network in each dimension, each packet requires at most $1/2 * (x + y + z)$ hops to reach its destination. On the average, a randomly generated packet will require $1/4 * (x + y + z)$ hops. Using this

information it became fairly easy to calculate the approximate number of total events. If m is the total number of messages, the expected number of events is close to $m * [1 + 1/4 * (x + y + z)]$.

With the large numbers of real messages being sent between physical nodes during our simulation runs, and a small amount of computation used to process each message, we did not expect to see good speedup figures, and have focused mainly on the scaling performance of the simulation. Nevertheless we did a small series of speedup tests, and were pleasantly surprised to obtain a 3.9 speedup when moving one of our simulations from 4 physical processors to 16 processors. For the speedup test we simulated 4096 Blue Gene nodes, each of which injected 100 packets.

The smallest simulation we used for the scaling tests ran on 8 physical processors and simulated 4K Blue Gene nodes, in a 16 x 16 x 16 torus. We injected 100 packets per simulated node, for a total of 400K packets. Including injection events and receive events, the simulation handled over 5 million events. We scaled this up in 5 steps to 64K nodes simulated on 128 physical nodes producing a total of more than 200 million events (see table 1).

We were quite pleased with the way the SPEEDES simulation scaled. The scaling performance was within about 10% of ideal, except for the 128 node case. Because the latter used all available nodes of the Origin, one of the processors was busy running some of the standard O/S tasks as well as our simulation, and we believe this caused the deviation shown in figure 1.

Conclusions and Future Work

SPEEDES has shown itself to be a valuable tool for the purposes of simulating a large computer network. It has shown itself capable of scaling up to handle the large number of objects and events required by this simulation. It is effective in its use of parallelism, both in handling large problem sizes, and in lessening the time required for the simulation.

Our main task for the near term is to increase the fidelity of the model in terms of the arbitration scheme that it uses as well as its adaptive routing techniques. Once that is completed, we will repeat the performance measurements previously done. Another part of our effort will be to add packet transit information into each packet as it is routed. This will enable the gathering of collective packet throughput statistics, so that we can better understand how the network operates, and whether there may be bottlenecks in certain areas.

Another team is working in parallel to produce characterizations of workloads based on specific science applications. This workload information will be used to drive our simulation to show how the system might operate under realistic applications.

With our long experience in developing simulations, we are particularly interested in understanding what optimistic techniques give the best performance for a given

simulation. With the ability SPEEDES offers to run the same simulation under different time management techniques, we can directly measure the effect on performance. We are particularly interested in doing this for those simulations which have unbalanced workloads.

SPEEDES has assisted us in this effort by proving to be an efficient framework for running parallel discrete event simulations.

Acknowledgment

This work was done by the Jet Propulsion Laboratory under contract with the California Institute of Technology.

References

1. Fujimoto, Richard M., "Parallel Discrete Event Simulation." *Communications of the ACM*, Vol. 33, No. 10 (October 1990), pp. 30–53.
2. Jefferson, David R., "Virtual Time." *ACM Trans. Prog. Lang. and Syst.* 7, 3 (July 1985), pp. 404-425.

Configuration	BG/L nodes	Physical Processors	Total Injected Packets	Average hops/pkt
16x16x16	4K	8	400K	12
16x16x32	8K	16	800K	16
16x32x32	16K	32	1.6M	20
32x32x32	32K	64	3.2M	24
32x32x64	64K	128	6.4M	32

Table 1. Benchmark Problem Sizes

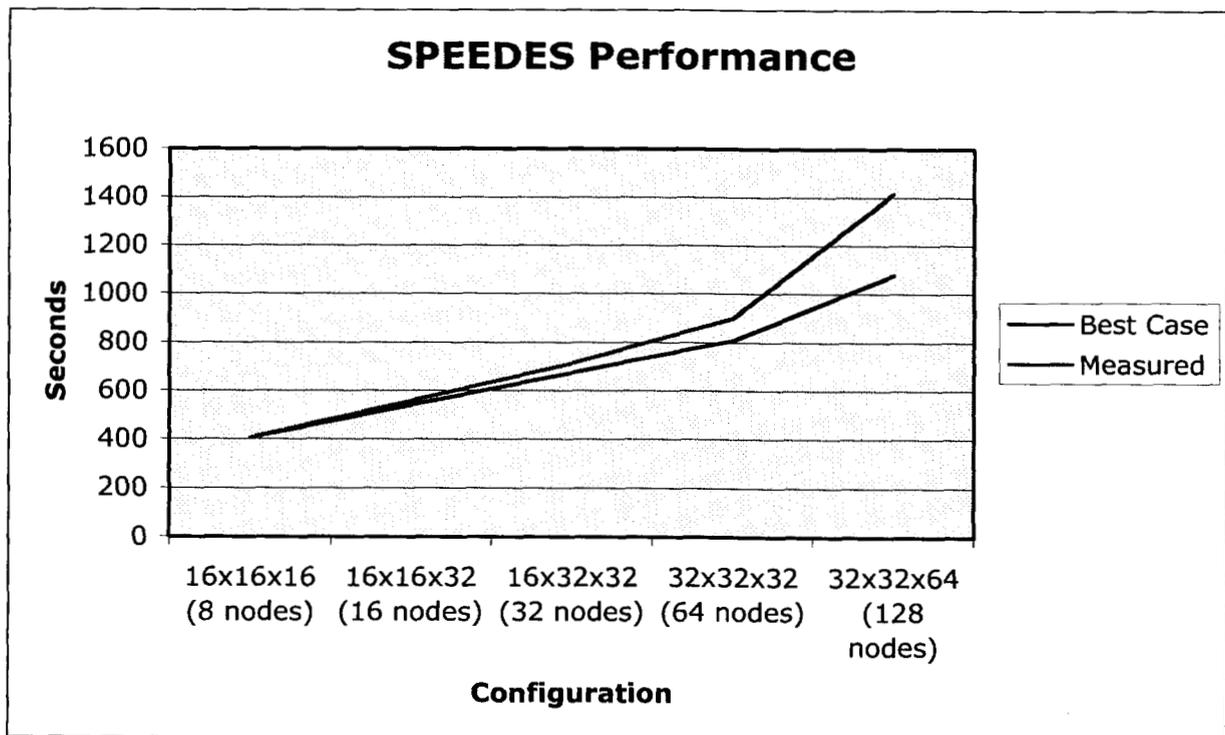


Figure 1. Scaling Results