Development of Naturally Fault Tolerant Algorithms for Computing on 100,000 Processors

Al Geist
Christian Engelmann
Oak Ridge National Laboratory
www.csm.ornl.gov/~geist

Abstract

This paper describes ongoing research at Oak Ridge National Laboratory into the issues and potential problems of algorithm scalability to 100,000 processor systems. Such massively parallel computers are projected to be needed to reach a petaflops computational speed before 2010. And to make such hypothetical machines a reality, IBM Research has begun developing a computer named "BlueGene" that could have up to 65,536 processor chips in the 2005 time frame. A key issue is how to effectively utilize a machine with 100,000 processors. Scientific algorithms have shown poor scalability on 10,000 processor systems that exist today. In this paper we define a new term called super-scalable algorithms, which have the property of natural fault tolerance, then go on to show that such algorithms do exist for scientific applications. Finally, we describe a 100,000 processor simulator we have developed to test the new algorithms.

Introduction

If computing power stays on the track of Moore's Law, then by 2010 the largest computers in the world will be in the petaflops range. Such computers will be enormous and require a rethinking of how scientific applications should be written to exploit these machines. It is possible to reach a petaflop even quicker if the parallelism of the machine is increased well beyond the few thousand processors used today. It would require on the order of 100,000 processors to reach a petaflop in the 2007 time frame. With the exception of embarrassingly parallel applications such as SETI@HOME [1], there is no experience with developing scientific applications on this scale. Where we do have experience, which is with 10,000 processor machines such as ASCI White [2], the efficiency of scientific applications can be as low as 1%. This means that the effective power extracted from the 10,000 processor machine was equal to 100 processors. Amdahl's Law shows how efficiency drops off as the number of processors increases. The danger for 100,000 processor machines is that the peak power might be in the petaflops range but the effective power may be only a couple of teraflops. To avoid such a problem it is important to start now and try to develop scientific algorithms that can effectively utilize 100,000 processor machines. This will include tuning and modifications to existing solution approaches as well as the development of new approaches to scientific problems.

A critical issue with machines of this size is the mean time between failures. Projecting from our existing supercomputers, a 100,000 processor supercomputer with all its

associated support systems could see a failure every few minutes. Applications today typically deal with faults by writing out checkpoints periodically. If a fault occurs then all the processes are stopped and the job is reloaded from the last checkpoint. With a 100,000 processor machine checkpoint/restart may not be an effective utilization of the resources. Does it make sense to kill 99,999 processes because one has failed? Looking forward we can see that at some point the time to do a checkpoint or a restart will be longer than the time until the next failure. When this happens the classical checkpoint/restart solution becomes completely impractical as a means to deal with failure. This paper explores the notion of naturally fault tolerant algorithms, which have the mathematical properties that they get the correct answer despite the occurrence of faults in the system.

The paper is organized as follows. The second section describes a project at IBM Research to build a computer with 64,000 processor chips called BlueGene [3]. In the third section we define what we mean by scale invariant and naturally fault tolerant algorithms. In section four we describe how algorithms that require local information such as finite difference and finite element algorithms can be reformulated to have natural fault tolerance. We illustrate this with a finite difference example. In section five we describe how algorithms that require global information can be formulated to have natural fault tolerance. We illustrate this with a global maximum example. In section six we describe a 100,000 processor simulator we have developed at Oak Ridge National Laboratory (ORNL) to test scaling and natural fault tolerant applications. The final section summarizes the results.

IBM BlueGene System

In December 1999 IBM announced a new research initiative called BlueGene with the goal of building a petascale computer to simulate protein folding and other Life Science problems. Over the years the initiative has evolved and the machine being designed today is called BlueGene/L. BlueGene/L is a joint research partnership with Lawrence Livermore National Laboratory through the Department of Energy (DOE) ASCI PathForward program [4]. In 2001 IBM signed a cooperative research and development agreement with Oak Ridge National Laboratory for development of computational biology applications for BlueGene as well as to explore the concepts of self-healing and naturally fault tolerant applications. IBM is particularly interesting in our research in a fault tolerant MPI implementation, FT-MPI [5]. The BlueGene project involves researchers at several DOE labs and universities. For example Argonne National Laboratory is helping develop MPI for BlueGene/L and is also looking at the issues of fault tolerance for MPI [6]. CalTech is developing a BlueGene interconnect simulator.

A major design goal of the BlueGene initiative is to produce a computer with a performance/cost ratio that was 5 to 10 times better than other supercomputers. To do this IBM leveraged commodity embedded processor technology. The embedded technology also produces much less heat so IBM could use a much higher density of processors. The disadvantage of using embedded processors is that their individual performance is

relatively low. This leads to using a large number of processors to reach a given overall performance level.

BlueGene/L is based on IBM's PowerPC embedded processor and system-on-a-chip technology that allows compute processor, communications processor, three cache levels, 4MB of embedded DRAM, and three high-speed interconnection networks to be placed on a single chip. Each compute processor is expected to have a performance of 2.8 Gigaflops. There are 65,536 such chips in a full size BlueGene/L system. Together they will provide a peak of 180 Teraflops.

The communications processor is identical to the compute processor with the only difference being the software running on it. If communication performance is not an issue for a particular application, then the communication processor can be configured as a second compute processor. In this configuration the BlueGene/L computer would have a peak performance of 360 Teraflops.

With so many processors in BlueGene/L the interconnection network becomes critical for effectively utilizing the tremendous processing power available. BlueGene incorporates not one but three separate interconnection networks. The first is a 64x32x32 three-dimensional torus that is used for general-purpose point-to-point communication. The torus links from each node can transfer data at 2.8 Gb/s, which leads to a bisection bandwidth of 2.8 TB/s for the torus network. The second network is a global tree that is designed to handle one-to-all and all-to-all functionality. Broadcast latency is predicted to be about 1 µsec with a bandwidth of 1.4 GB/s from a single node to all other nodes. The third network is Ethernet and is used for host control, booting, and diagnostics.

IBM's BlueGene/L represents a real manifestation of a 100,000 processor supercomputer in the 2005 time frame. The algorithms and simulator described in this paper have a direct bearing on the utility of such a machine.

Super-Scalable Algorithms

Our research into computing on 100,000 processors shows that there are two serious impediments to taking a "business as usual" approach to applications for such large machines. The first is Amdahl's Law and the need to reduce the serial fraction to a point where the application can get a reasonable efficiency. The second is the high probability of multiple node failures during a single application run and how the applications can recover. We are working to establish a theoretical foundation for a new class of algorithms called super-scalable algorithms that have the properties of scale invariance and natural fault tolerance.

We use the term scale invariance to mean that the individual tasks in the larger parallel job have a fixed number of other tasks they communicate with independent of the number of tasks in the application. A simple example is a finite difference algorithm. The number of neighbor tasks is fixed by the stencil used and independent of the total number of tasks in the overall problem. Another simple example is a binary tree where each task talks to

only three other tasks. With scale invariance individual tasks do not have to be concerned about failures throughout the system unless these failures happen to affect one of their neighbors. Conversely, dynamically adding replacement or additional tasks to a problem can be ignored by tasks not communicating with these new tasks.

As defined, scale invariance does not guarantee that an algorithm will achieve high efficiency on a 100,000 processor system. Efficiency is based on the serial fraction of a parallel algorithm and depends not only on a particular application but also on the hardware it runs on. Factors like I/O and cache misses can quickly drive efficiency down even if the best known algorithms are being used for the solution.

Scale invariance also doesn't make an algorithm fault tolerant. Most parallel algorithms designed today will deadlock, or worse—calculate the wrong answer, if one or more of the tasks fails during a computation. What scale invariance does do is isolate the failure and not make it a property of the total number of tasks. Fault tolerance can then be handled locally by self healing or natural fault tolerance.

We say a parallel algorithm has natural fault tolerance if it is able to get the correct answer despite the failure of some tasks during the calculation. It is not that the calculations are taken over by other tasks, but rather that the nature of the algorithm is that there is natural compensation for the lost information. For example, an iterative algorithm may require more iterations to converge but it still converges despite lost information.

The number of tasks that can fail and still get the correct answer is problem dependent and still an open research question. Our assumption is that the number of tasks lost during an application run will be a small fraction of the overall number of tasks. While the ideal case is that no tasks fail during a computation, even if 100 tasks failed on a 100,000 task application, this amounts to only one tenth of one percent of the tasks failing. This is the range of failures we have been considering. Experience with existing supercomputers indicates that the majority of processor failures are due to software problems either in the OS or runtime system. Additional failures may occur simply due to bugs in the application code.

Today if 100 processors failed every time an application was run, it would be time to send that computer back to the manufacturer. For petascale supercomputers this may not be the case. Faulty parts may just be replaced or rebooted while the machine is running. The ability to complete a computation despite failures without stopping the entire machine is an emerging need for these future computers.

Natural fault tolerance is a restrictive requirement on algorithms and it was not clear when we began our research that anything other than the most trivial applications would be able to meet the definitions.

The most trivial parallel applications have all independent tasks. They are scale invariant since each task communicates only to send back its answer. By using a bag-of-tasks

programming paradigm where the independent tasks are farmed out, and if one fails a new one is farmed out in its place, the application can have natural fault tolerance. Task farming is a widely used technique today. SETI@HOME [1] uses it for Internet computing. Condor [7,8] uses it for Grid computing. Both of these examples incorporate on the fly fault tolerance by task replacement.

SETI has gone one step further and has had to address validation of the answers because users were falsifying the answers being sent back. SETI's solution is to redundantly calculate everything twice and where there is disagreement calculate the solution one or more additional times. For most large supercomputer applications having to calculate the same answer multiple times may not be an effective use of the resource. So the approach of multiple computer runs to validate may not be practical or necessary on a petascale computer.

We have developed naturally fault tolerant algorithms for non-trivial algorithms and describe them in the next two sections. But we are also aware that there will always be a large number of algorithms where natural fault tolerance is not possible. One such example is FFT. While there are many ways to do Fourier transform that are naturally fault tolerant, the nature of fast Fourier transform is that every calculated value is crucial to the correct solution. Thus failures cannot be ignored. In the cases when a naturally fault tolerant algorithm cannot be found, we have investigated adapting traditional detect and recover methods to 100,000 processor machines.

There are three steps to traditional fault tolerance. First is detection. If the failure isn't detected, then recovery is never initiated. Second is notification. At a minimum the runtime must be notified that a failure has occurred in the application. The tasks in the application may also need to be notified so that the application can determine if the nature of the failure is recoverable and how to proceed.

Recovery involves two steps—previously saving a copy of state somewhere other than the affected node and restoring that state on the same or another node after a failure. We are presently investigating having each task periodically distribute its state to the tasks it communicates with and these tasks store their part of the state in their local memory. This is similar to the work of Li and Plank with diskless checkpointing [9]. In-memory checkpointing provides the opportunity for very fast state recovery and avoids the problem that disk storage may be very far away from a given processor in a 100,000 processor system.

The state is distributed using the same techniques as RAID storage. This allows recovery even if one of the neighboring tasks also fails at the same time as the task in question. By using neighbors the checkpoint step is much more parallel than trying to writing to disks. So far these ideas have not been tried on 100,000 processor simulators.

Failure recovery can be avoided if the algorithm is naturally fault tolerant. However, failure detection and notification are still needed to inform the algorithm to adapt. In the next two sections we describe naturally fault tolerant algorithms we have developed in

two broad problem areas. The first is where the problem can be formulated as some function of a local volume. Such problems include finite difference and finite element applications as well as many others. The second is where the problem requires global information. Such problems include finding the global maximum or minimum and are often used to determine if an iterative algorithm has converged.

Local Information

To demonstrate that naturally fault tolerant scientific applications exist that are not embarrassingly parallel. We started by looking at parallel applications where the tasks require only information from a local region. Examples include finite difference and finite element solutions to differential equations. Requiring information from a small number of nearby tasks is the next step up in difficulty from requiring only information from one's own memory (independent task).

This next step brings some significant changes. First the tasks must be aware of and be able to communicate with at least a small set of other tasks. Second the tasks need to be able to synchronize with each other—either implicitly through message passing or explicitly so that the parallel computation can proceed in the correct order. Third the tasks should be able to deal with the failure of one or more of the tasks they communicate with in a manner consistent with the computation being performed.

The approach we took combines two ideas: chaotic relaxation [10,11] and meshless methods[12]. The first is to avoid having to synchronize across all the nodes between iterations. The second is to be able to adapt to faults without requiring a replacement node to fill in the hole in the grid. Together these two ideas form the basis for a naturally fault tolerant algorithm.

We formulated the application using meshless methods rather than a fixed mesh with a finite difference stencil of neighbors. Standard difference equations and their error properties break down when information from one of the stencil nodes is left out. There are several ways to formulate meshless solution methods. We use the approach of randomly placing the nodes across the problem space with a uniform distribution. Each node is then given a list of m nodes locally around it including their location so that distance and direction to each of the neighbors can be calculated. The value of m is chosen by the application developer based on the nature of the calculation being performed at each node. The information is there so that these values can be calculated, usually once at setup, if needed. Figure 1 shows a 2-D meshless layout. The distance and direction are not needed for all applications, but are crucial for weighting the data coming from the neighbors in some applications. In the example given in this section the distance is required but direction is not needed. In the next section we will show an application where neither distance nor direction are needed.

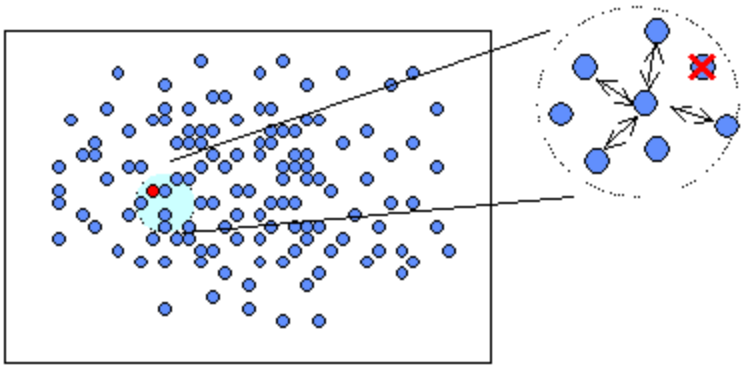

Figure 1. Meshless formulation of 2-D finite difference application using a uniform distribution of nodes and a user selected number of local neighbors.

A potential problem is how to synchronize 100,000 nodes particularly when some may be very distant relative to the nearby nodes. One solution as seen in the BlueGene design is to have separate hardware (the tree network) to allow fast dissemination of information to all the nodes. If a hardware solution is not available then some O(log(p)) software solution is an alternative. Both of these solutions have problems when node failures occur either before or during the synchronization.

To demonstrate a natural fault tolerance finite difference algorithm we chose the 2-D Poisson Equation as the problem. For this problem we use chaotic relaxation instead of synchronizing between iterations. In chaotic relaxation all nodes update their values asynchronously and send out their results as soon as they are calculated.

Chaotic relaxation was investigated in the 1970's in the context of parallel computation. Researchers found that the conditions under which chaotic relaxation converges is more restrictive than standard methods and it never became popular. For 100,000 processor computations it may be time to once again look at this iteration-free method. When failures and failure recovery are factored into the solution time, chaotic relaxation has some attractive recovery properties. First, recovery can be done independently and locally by just the nodes that communicate with the failed node. Second, the lost information (state) of the failed node does not need to be backed up or recovered. The calculations can be formulated to proceed and converge despite failed nodes.

Convergence and speed of convergence are a concern when using meshless methods or chaotic relaxation, and even more so when loosing the information calculated by failing nodes. We have run a number of experiments on a 100.000 processor simulator. For the level of failure we are interested in (100 failures or less during a run) the perturbation caused by failure of one tenth of one percent of the nodes is not perceptible in the rate of convergence or the accuracy of the results. We have run the algorithm with much higher levels of failure to observe the behavior and as expected we found both accuracy and speed of convergence degrades. More studies are needed to parameterize the affects of high failure levels on solution speed.

The finite difference solution to the Poisson equation can be thought of as an averaging of values in nearby nodes. The natural fault tolerance implemented in our algorithm proceeds in the following way, when one of the nearby nodes fails the calculation adapts by simply averaging over a smaller number of nearby nodes as seen in Figure 1. Each node runs the following algorithm.

Natural Fault Tolerant Finite Difference

Compute nodes average their values based on the values and distances of their neighbors. New values are multicast to neighbors if the change exceeds epsilon. Boundary nodes simply multicast new boundary values to neighbors.

```
procedure main:
  initialize constant epsilon
  initialize stored value
  initialize stored neighbor values
  compute and store distances to neighbors
  do forever
    if not boundary node
      receive value from any neighbor
      store neighbor value
      compute new value using neighbor values and distances
```

The new value $v = \frac{\sum_{n=1}^{m} \frac{v_n}{d_n}}{\sum_{n=1}^{m} \frac{1}{d_n}}$, *where* v_n *is the value of neighbor* n, d_n *is the distance to neighbor* n, *and* m *is the number of neighbors.*

```
      if local value differs from new value by more than epsilon
        set local value to new value
        multicast new value to all neighbors
      end if
    else
      get new boundary value
      multicast new boundary value to all neighbors
    end if
  end do
end
```

A node stops sending values to a neighbor and deletes the stored neighbor value and distance if the neighbor fails.

```
procedure failure notification:
  identify failed neighbor
  stop sending values to failed neighbor
  delete stored value and distance of failed neighbor
end
```

Global Information

Once it was established that naturally fault tolerant algorithms can exist when only communication with local neighbors is required, we expanded our search to see if naturally fault tolerant algorithms could exist when global information is required. There are many physics problems where global information is needed, for example those involving gravity and radiative transport often require the need for the calculation of long-range forces. Global information is also needed in most iterative methods to determine if convergence has been achieved.

The restriction we impose on super-scalable algorithms is that tasks can communicate with only a constant number of neighbors. Not with a f(p) number of neighbors such as log(p) neighbors where p is the total number of processors in the system. The first thing to observe is that a tree network is the standard communication pattern used for determining global information. But a tree will not work if node failures occur because one or more branches of the tree become isolated and will not get the correct answer.

Eliminating tree topologies, there are still several node topologies that could be good fits to global information problems.

- A node's neighbor set is chosen based on the physics of the underlying problem, for example where neighbors are chosen from all the nodes but using a distribution function like $1/r^2$ radiating out from each node.
- A node's neighbor set is chosen based on a regular fixed pattern such as a 3-D mesh or hypercube.
- A node's neighbor set is chosen randomly from all the nodes with a uniform distribution.

The properties needed are that the graph has to be connected to start with. It can't have isolated islands of nodes. Even more restrictive there must be enough degrees in the graph so that a single node failure does not isolate a portion of the graph and that the probability of several random node failures creating an isolated region should be very remote. It is obvious that with enough node failures isolation is inevitable so we continue to limit ourselves to the case of only about 100 of the 100,000 processors fails during the computation.

We chose global maximum as an example problem. The problem was stated as follows: Each node starts up and generates a random number. The proposed algorithm is run and while running any node or small set of nodes can be killed. When the algorithm finishes all the nodes that are still alive know the maximum value of the original set of random numbers. Special failure cases exist, for example, if the node with the maximum value is killed before it has a chance to send this value to any other nodes, then the algorithm finds the second largest random value.

Several communication topologies were tried for this problem and the one we found that had the most resilient behavior over a wide range of failure conditions was a random directed graph. Each node was given a random list of neighbor nodes to send to and a separate random list to receive from. The nodes in the lists were uniformly distributed

across all the nodes. We tried list sizes as small as four neighbors and as large as ten neighbors. Both worked and were able to tolerate much higher levels of node failure than the one tenth of one percent level.

As expected the more connected ten neighbor case converged faster. The information propagates across the system at a speed approximately equal to
Log $_{\text{base \#neighbors}}$ (p), where p is the total number of nodes.

Because we use a random directed graph, there is a probability that the graph is not connected or has a node whose degree will isolate it with a single node failure. But the properties of the constructed graph are such that the probability that either of these cases being true starts out very small and is driven rapidly towards zero as the number of neighbors increases. Like modern probabilistic prime testing algorithms, the probability of an incorrect answer can be made arbitrarily small.

What follows is the psuedo-code for the naturally fault tolerance global maximum algorithm.

Natural Fault Tolerant Global Max

All nodes compute their max and multicast it to their neighbors. If a received max is greater than the stored max, the received one overwrites the stored one and the neighbors receive an update.

```
procedure main:
  compute local max
  multicast local max to all neighbors
  do forever
    receive max from any neighbor
    if local max less than neighbor max
      set local max to neighbor max
      multicast local max to all neighbors
    end if
  end do
end
```

A node stops sending max values to a neighbor if the neighbor fails.

```
procedure failure notification:
  identify failed neighbor
  stop sending max to failed neighbor
end
```

100,000 Processor Simulator

While theoretical analysis of naturally fault tolerant algorithms can give some insight into convergence or the probability of achieving the right answer, there is a lot of practical information that can be achieved by coding up an algorithm and subjecting it to a variety

of different failure situations. Since BlueGene hardware will not be available for a couple more years, we needed a simulator. Several BlueGene simulators are under development around the world. Each is focusing on a particular aspect of the architecture. For example, IBM Research has developed an instruction-level simulator for the BlueGene chip but due to its overhead it is only able to simulate a few nodes. Cal Tech is developing a BlueGene network simulator [13] that takes as input an MPI communication trace produced by running an application on some other parallel computer. This trace is then run through the simulator and the output is an analysis of the performance of the BlueGene network for this same communication pattern. There is a BlueGene simulator developed on top of CHARM++ [14] with several of the features we needed, but we were unable to get it to scale beyond 30,000 nodes without crashing. Our early attempts building a 100,000 processor simulator ran into similar problems. What we needed was a tool that could simulate 100,000 processors and also simulate different failure conditions across the processors. Finding none, we created such a tool.

In designing the simulator one of our goals was to create a tool that could be used by scientists developing applications for machines with an order of magnitude greater number of processors than exist today. Firstly, to get them to think about reformulating their application to scale to 100,000 processors and secondly, to consider what their application should do if there are failures across the machine while the application is running.

The simulator is designed to test algorithms at very high scale and provide a platform to develop applications. It is not instrumented to give performance estimates or analysis of the applications for a particular machine.

The simulator can handle multiple languages and run on different operating systems. We have run it under both Linux and Windows. The simulator itself is written in Java and runs on any system that supports a Java Virtual Machine. We supply application examples written in C, Fortran, and Java with the simulator to facilitate scientists developing their own applications for the simulator.

The number of nodes that can be simulated depends on the size of the application being simulated and the power of the hardware the simulator is running on. The simulator is itself a parallel application and can run across a Linux cluster. On a 2 GHz Windows laptop we have simulated 10,000 nodes of a small application. Using a 32 processor, large-memory Linux cluster we have simulated nearly a half a million nodes running the naturally fault tolerant algorithms described earlier in this paper.

The simulator user interface for the finite difference example is shown in figure 2. The outside sliders allow the user to dynamically change the boundary conditions. The colored dots, which can be adjusted in size for best viewing, represent the running tasks in the simulator. Each is context switched into the simulator as a program, data, and communication lists run for a period of time under the OS available, for example Linux, then switched back out. The color of the dot is controlled by a call that can be placed inside the user's application to display some value that is relevant to the progress of the

given application. In the finite difference example, the color represents the present value calculated at each task based on the boundary conditions. Clicking on a dot in the user interface brings up an information window about the selected task as well as an option to disable (kill) this task. What information is displayed in this window is under the control of the application developer. Finally, under the "Control" menu is the option for selecting a region of dots and disabling all or a percentage of random tasks in the region.

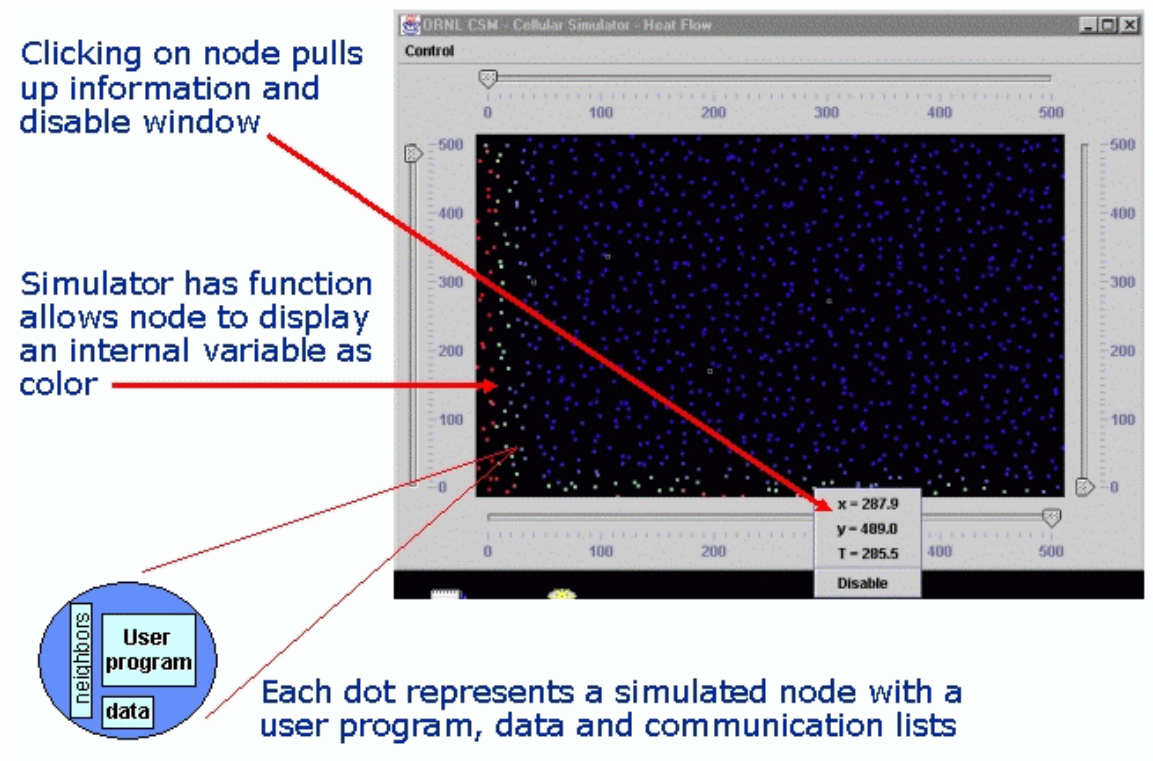

Figure 2. User interface to the ORNL 100,000 processor simulator.

The simulator has a configurable network topology among the nodes, which the user sets up before running an application. Each simulated node has a list of nodes that it can send to and a separate list of nodes that it can receive from. When these lists are the same then a bi-directional network is defined. When they are different a uni-directional network is simulated. The lists can be anything from a random selection of nodes to a very specific pattern defining a hypercube or some other network topology. Thus the simulator is not specific to the BlueGene topologies and it can be used to investigate any number of alternatives. To simplify topology configuration for the user, the simulator has a configuration panel that allows the selection of the number of nodes to simulate and a growing list of predefined topologies. Topology choices presently include: torus and mesh, where the user also specifies the dimension of the topology, and topology choices of random nearest neighbors, and globally random neighbors, where the user specifies the number of neighbors desired. For example the user can specify a 50,000 node 3-D mesh or 64,000 node locally random interconnect as used in the finite difference algorithm given above. The topology specification is a module and the user can add to this list.

The simulator has a number of built in failure modes that the user can specify. It allows killing of a selected node, block of nodes, or a random percent of nodes in a specified region. The failures are presently interactively initiated i.e. the user clicks on a node and kills it, or selects a region and 1% of the nodes in this region dies. Although possible, the simulator presently does not do time-based failures, for example, produce a random failure every 5 minutes.

Summary

This paper has described the first steps in ongoing research at Oak Ridge National Laboratory into the issues and potential problems of algorithm scalability to 100,000 processor systems. A key issue is how to effectively utilize a machine with 100,000 processors. Scientific algorithms have shown poor scalability on 10,000 processor systems that exist today. In this paper we defined a new term called super-scalable algorithms, which have the property of natural fault tolerance, which means that the algorithm gets the correct answer despite the failure of nodes. We describe the construction of super-scalable algorithms for the case where the scientific calculation can be described as some function of the local region and show the pseudo-code for when this function is the average of the local region. We also describe the construction of a O(log(p)) global maximum algorithm that illustrates how global operation algorithms can be constructed using a random directed graph to get the correct answer despite node failure. Finally, we describe a 100,000 processor simulator we have developed to test the new algorithms at high scale and under various failure scenarios. For future work we are looking at developing naturally fault tolerant algorithms for FFT and multigrid. These will be the subject of future papers.

References

[1] SETI@HOME Project, online at http://setiathome.ssl.berkeley.edu, Space Sciences Laboratory, University of California Berkeley, USA.

[2] ASCII Program, online at http://www.llnl.gov/asci, Lawrence Livermore National Laboratory, USA.

[3] N. R. Adiga, et al, “An Overview of the BlueGene/L Supercomputer”, *Proceedings of SC2002, also IBM research report RC22570 (W0209-033)*, November 2002.

[4] ASCII Blue Gene/L Project, online at http://www.llnl.gov/asci/platforms/bluegenel, Lawrence Livermore National Laboratory, USA.

[5] G. Fagg, A. Bukovsky, and J. Dongarra, “Harness and Fault Tolerant MPI”, Parallel Computing, vol . 27, num. 11, pp 1479-1495, October 2001

[6] William Gropp and Ewing Lusk, “Fault Tolerance in MPI Programs”, *Proceedings of the Cluster Computing and Grid Systems Conference*, December 2002.

[7] Condor Project, online at http://www.cs.wisc.edu/condor, Computer Sciences Department, University of Wisconsin, USA.

[8] Jim Basney and Miron Livny, "Deploying a High Throughput Computing Cluster", *High Performance Cluster Computing*, Rajkumar Buyya, Editor, Vol. 1, Chapter 5, Prentice Hall PTR, May 1999.

[9] J. S. Plank, K. Li, and M. A. Puening, "Diskless Checkpointing", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, No. 10, pp. 972-986, October 1998.

[10] D. Chazan, and M. Miranker, "Chaotic Relaxation", *Linear Algebra and its Applications 2*, pp199-222, 1969.

[11] G. M. Baudet, "Asynchronous Iterative Methods for Multiprocessors", *Journal of the ACM*, Volume 25, Issue 2, pp. 226-244, April 1978.

[12] G. R. Liu, "Mesh Free Methods: Moving beyond the Finite Element Method", 1st Edition, CRC Press, July 2002.

[13] Personal communication, "BlueGene/LPerformance Prediction: Computational Experiments, Simulation, and Analysis", Center of Advanced Computing Research, California Institute of Technology, USA.

[14] G. Zheng, A. K. Singla, J. M. Unger, and L. V. Kale, "A Parallel-Object Programming Model for Petaflops Machines and Blue Gene/Cyclops", *Proceedings of IPDPS 2002*, April 2002.