Emulating PetaFLOPS Machines and Blue Gene

Neelam Saboo
Arun Kumar Singla
Joshua Mostkoff Unger
Laxmikant V. Kalé
Dept. of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Ave.
Urbana, IL 61801
{saboo, asingla, unger1, kale}@cs.uiuc.edu

Abstract

PetaFLOPS-class computers, based on the current or foreseeable CMOS generation, appear to be feasible in the near future. An emulator for a petaFLOPS-class programming environment is necessary to facilitate offline development and debugging of applications, and exploration of programming models. Such an emulator must be able to run on large traditional parallel machines. This paper describes the design and implementation of an emulator for a class of petaFLOPS machines. The machine parameters can be varied to cover a variety of possible architectures within this class, although our current implementation is influenced by (and is targeted to emulate) an initial design of the Blue Gene Machine being developed by IBM. Our implementation is based on Charm++, an object-based message-driven parallel execution model, which allows emulation of multiple Blue Gene nodes to a single physical processor. We demonstrate the feasibility of our approach by emulating short million-processor programs on less than a hundred processors of the ASCI-Red machine.

1. Introduction

A petaFLOPS computer would be hundreds of times more powerful than the current largest parallel computer. Several techniques have been proposed for building such a powerful machine. Some of the designs call for extremely powerful (100 GFLOPS) processors based on superconducting technology. The class of designs that we focus on use current and foreseeable CMOS technology. It is reasonably clear that such machines, in the near future at least, will require a departure from the architectures of the current parallel supercomputers, which use few thousand commodity microprocessors. With the current technology, it would take around a million microprocessors to achieve a petaFLOPS performance. Clearly, power requirements and cost considerations alone preclude this option.

The class of machines of interest to us use a "processors-in-memory" design: the basic building block is a single chip that includes multiple processors as well as memory and interconnection routing logic. On such machines, the ratio of memory-to-processors will be substantially lower than the prevalent one. As the technology is assumed to be the current generation one, the number of processors will still have to be close to a million, but the number of chips will be much lower.

Using such a design, petaFLOPS performance will be reached within the next 2-3 years, especially since IBM has announced the Blue Gene project aimed at building such a machine. However, petaFLOPS machines will not be operational for a few years and once they are built access to them will be limited. Thus, an emulator for a petaFLOPS machine is needed to develop, and test the applications that will run on petaFLOPS computers, to experiment with alternative algorithms, and to design new programming models for them. Even after the machines are available, a programming environment emulator will be invaluable for offline debugging, testing, and possibly performance studies of applications.

We have developed an emulator to meet this goal of providing a programming environment for application development. A major challenge in building such an emulator is that of capacity: a single processor will not be able to emulate a program that is designed for a million processor system, mainly because of memory limits. So, the emulator must be a large traditional parallel computer itself. Our emulator is capable of utilizing machines with hundreds or even thousands of processors.

In this paper, we use IBM's Blue Gene machine (http://www.research.ibm.com/bluegene) and related architectural variants as concrete examples. The benchmark speed of petaFLOPS is very important to Blue Gene's goal of running a large scale Molecular Dynamics simulation on the order of tens of thousands of atoms. In a MD simulation, the forces that each atom exerts on the others and the resulting kinetics of each atom are calculated for discrete timesteps, usually on the order of a femtosecond (10^{-15} s). Even running at one petaFLOPS, the algorithm is so computationaly intensive that Blue Gene will have to run at its full capacity of over a million processors for a full year in order to reach its goal of studying the mechanisms and advancing the science of protein folding.

For concreteness, we next describe an initial design of Blue Gene machine. Our emulator is capable of modeling many architectural variants within this genre. Section 3 gives an overview of a low-level, generic programming environment that the emulator supports. The design of the emulator, along with an overview of the Charm++ system on which it is built is presented in Section 4. Issues involved in optimizing the performance of the emulation, along with some data demonstrating the efficacy of our current implementation is presented in the Section 5. We view the emulator as a first step on an ambitious research program aimed at emulating, and simulating petaFLOPS computers, and developing programming environments and applications for them. The future work planned is discussed in Section 6.

2. Blue Gene Architecture

IBM's Blue Gene, currently under development, represents a specific architectural design in the range of potential petaFLOPS-class computers. The computer's name, Blue Gene, reflects its application of studies of proteins and computationally intensive molecular dynamics algorithms. In this section we describe a possible plan for Blue Gene machine architecture based on information published on IBM's Blue Gene website [1]. Although the final Blue Gene architecture may differ from this since the design is still evolving, we think it still represents a valid member of the class of petaFLOPS machines we are interested in.

This class of high-performance computers is defined by a homogeneous collection of interconnected multi-processor nodes with a relatively low memory to processor ratio. Such a machine will lead to a substantial performance increase due to its larger number of nodes. The large number of nodes and communication latencies between them uncovers new problems in Operating Systems and software design for such machines. For example, the relevance of topology has again returned to the parallel computing field with computers like Blue Gene.

In order to achieve performance of one petaFLOPS such a machine may consist of over 1 million processors running over 8 million concurrent threads. A possible packaging for such a machine is shown in Figure 1 [1].

1. *Processor*: Each "processor" includes 8 processing elements(PEs), called thread units, each with its own integer ALU, and 64 registers. The PEs within a processor share two floating point units (a multiplier and an adder), an I-Cache and a load-store unit (See Figure 2). Each processor has its own (DRAM) memory and can also access local memory of other processors on the same chip.

2. *Node*: A node ("chip") has 25 processors (200 threads) connected in torus topology via a 64-bit local bus to give a total performance of 25 gigaflops (Figure 3a). The memory of each processor is available to any of the processors on the node, totalling a node's memory to 12.5 MB. The Input Buffer holds the incoming messages to a node. Any outgoing messages are stored in Output Buffer and are shuttled to a destination node by hardware. All message communication is asynchronous. Each node has six full duplex links and associated switches, connecting it to its immediate neighbours in the cube geometry (Figure 3b).

Beyond node level, the chips are physically organized into boards, racks and towers as shown in figure 1. However, logically the machine can be considered to be a $34 \times 34 \times 36$ three-dimensional grid of nodes for a total of about 40,000 nodes. With each node containing 25 processors (200 threads) the total is well over a million processors and 8 million threads with a theoretical peak performance of over a petaFLOPS.

Figure 2 shows a more detailed schematic of the Blue Gene processor. There is a processor-side instruction cache of about 32 KB with pre-fetch of 32 instructions. The 512 KB of memory associated with each processor has a memory-side cache. Because each processor on a node can access other processors' memory on the same node the memory-side cache eliminates synchronization problems. Each processor should really be considered to be 8 processing elements since eight threads run concurrently with round robin scheduling. Multiple independent threads per processor are supported for tolerating the memory latency. In one early estimate, it was calculated to be 10 cycles for a thread to access its local cache, 20 cycles for its local memory, 20 cycles for cache on remote memory(within the same chip), and 30 cycles for non-cache remote memory. While a thread is waiting for its memory fetch to complete, the other threads can do work. The threads on a processor share two floating point units: one for add and one for multiply. A floating point operation takes six cycles to complete. Each floating point unit is pipelined accepting one

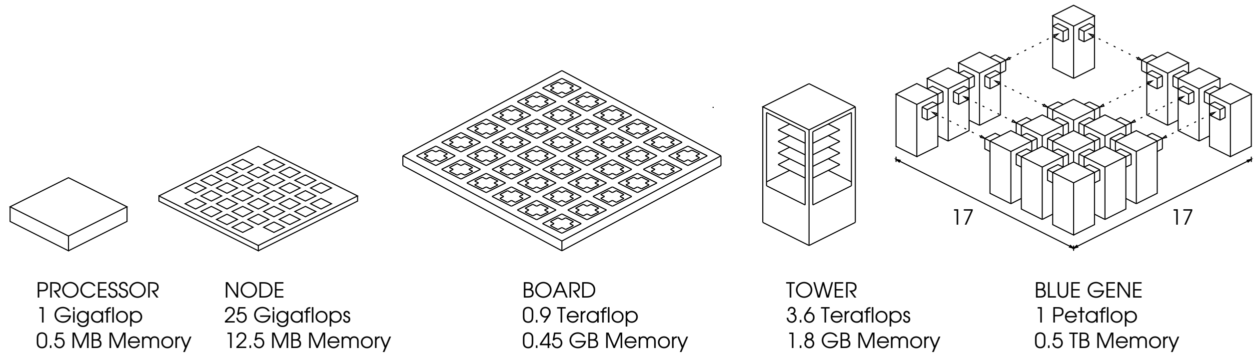

Figure 1. Five Steps to a PetaFLOPS Computer (picture based on data at IBM's website [1])

instruction each cycle. (All of these specific parameters are estimates.)

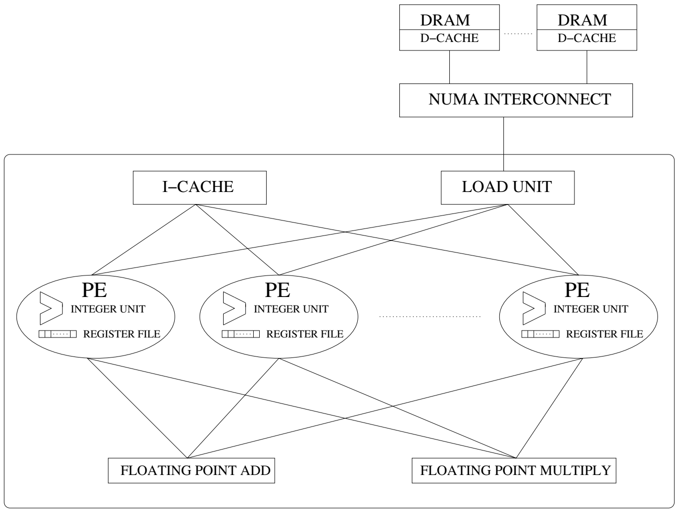

Figure 2. Blue Gene Processor, Processing Elements and NUMA Interconnect

The node architecture is shown in Figure 3. Twenty-five processor elements are connected in a 64-bit torus connectivity so each processor of the node can access memory on other processors as described above. Processors do not have direct access to the memory at other nodes. In order to communicate with other nodes, there is an input buffer and an output buffer where asynchronous packets are exchanged. Each packet is 128 bytes in length (although not all of the space is necessarily used) and each buffer can store 32 packets. If the input buffer is filled, the hardware will wait until the buffer is free before delivering the packet. If an output buffer is filled, the thread sending the message must wait. An outgoing packet has the basic format of an eight byte header with the Δx, Δy, Δz of the destination node followed by program-defined payload. Nodes are connected to their neighbors via full duplex links of 16 bit-width in each direction operating at 500 MHz for a transfer rate of one gigabyte per second. As a packet travels from node to node via this network, the packet header's Δx, Δy, and Δz is updated. Estimated latencies are int the order of five cycles per hop from node to node and 75 cycles when the packet turns a corner.

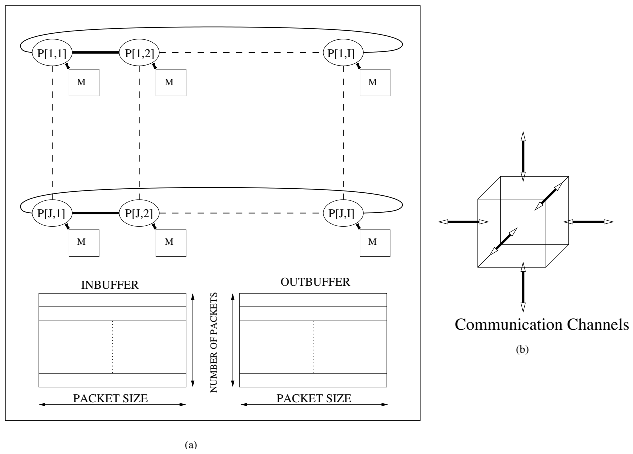

Figure 3. Blue Gene Node Architecture

3 The Programming Environment

We chose to support a low-level, but fairly general, API in the emulator. This was necessary, since more sophisticated parallel programming environments have not been designed yet for petaFLOPS machines. A low-level API allows one to build such programming environments on top of the emulator. We chose an API that mimics the Blue Gene low-level API, but is quite general to cover other architecture variations. Specifically, the API supports multiple instruction streams (threads) within a chip, one for each processing element. Each thread has its own stack. A simple access to the reliable communication layer is provided via calls that send short messages to other nodes, along with an index of the handler to be invoked the destination. The message length may be limited depending on the machine being emulated. For Blue Gene, it is limited to around 100 bytes.

In the model described so far, the threads on a processor communicate only via shared memory. Although this method is adequate to cover all forms of coordination, we decided to support an additional abstraction: that of a micro-task scheduler. The scheduler provides access to the common pool of work from which tasks may be scheduled on any individual thread.

The emulator supports the following API

- *BgNodeInit* - is called by the runtime system for initialization on each node, where application handlers are registered and computation is triggered by creating micro-tasks at the required nodes.

- *void registerHandler(int handlerID, BgHandler h)* - is invoked to register a handler with each node. Each handler has a globally unique identifier associated with it.

- *void addMessage(PacketMsg *msgPtr, int handlerID, int threadCategory)* - is called to create a micro-task.

- *void sendPacket(int x, int y, int z, PacketMsg *msgPtr, int handlerID, int threadCategory)* - is used to send a message to a node at location [x,y,z] in the node grid.

- *Utility functions* - In addition, it supports several utility functions that allow access to timers, the identity of the node and the processor on which the invoking thread is running, etc.

The above API has the advantage of being small yet complete. Other functions, such as "get", and "put", as well as high-level models can be built on top of this simple layer.

Application programming using this API is similar to that using other data-driven systems, such as our own Converse [2] and Charm++ [3] systems, as well as others such as active messages [4]. The difference lies primarily in the message-length limitation. In the version supported by our emulator, messages are only 128 bytes in length, so functions must break their data into discrete parts of at most 128 bytes each. The handler function that receives the discrete parts must stage the data until the complete data is ready, and then call a separate function to process the data. Alternatively, they may pipeline the computation itself, so some computation gets done every time a packet arrives.

4 Emulator design

For emulation, it is necessary to model the petaFLOPS machine using a traditional parallel programming system. Notice that the emulation is feasible only because of the low memory-to-processor ratio on the petaFLOPS machine simulated. For example, Blue Gene is likely to have about 0.5 TeraBytes of total memory. So, even if all the memory is being used by an application, it can be emulated (for example) on 2000 processors of a traditional parallel machine with 256 MB each. A parallel emulation poses another potential problem: messages may be delivered in a different order compared with their delivery on the emulated machine. Therefore, the application must be written so as to handle out of order message delivery. This is not a significant obstacle as many parallel programming paradigms and their runtime systems already handle this behavior.

Charm++, the parallel object system developed at University of Illinois, is well-suited for this purpose. In the following subsection, we briefly summarize features of Charm++. The architecture of the emulator is described next.

4.1 Charm++: Parallel Programming Model

Charm++, used to emulate the functional architecture described above, is described in detail in [3]. For completeness we include a brief summary of its features taken from [5]. Charm++ implements an object-based message-driven execution model [6], where *entry methods* of objects are invoked by messages received on the processor where the object resides. Object methods are executed non-preemptively and new messages are retrieved from a scheduler queue when an entry method is completed. This simplifies programming because applications need not worry about inconsistencies that might arise in a preemptive environment.

There are three types of objects in Charm++. *Chares* are single instances of objects. Once a chare is created, it may send its chare ID to other objects, who may then use that ID to invoke methods on that chare. The second type of Charm++ objects are *Groups*. Creation of an object group results in the creation of one object instance on each processor, identified by a group ID. Individual objects in an object group are accessed using the pair of group ID and processor number. Other objects may invoke entry methods on any particular processor of an object group, or may broadcast method invocations to every member of the group. The third type of objects, *object arrays* [7], are arbitrarily-sized sets of objects. The size of an array can either be defined during array creation or changed during program execution. Array elements communicate via remote method invocation, using their *array ID* and element index.

Charm++ objects can migrate from one processor to another during run-time without requiring other objects to know their physical location explicitly. The run-time system forwards messages to objects as necessary. A load balancer controls the migration of objects to optimize the usage of all processors.

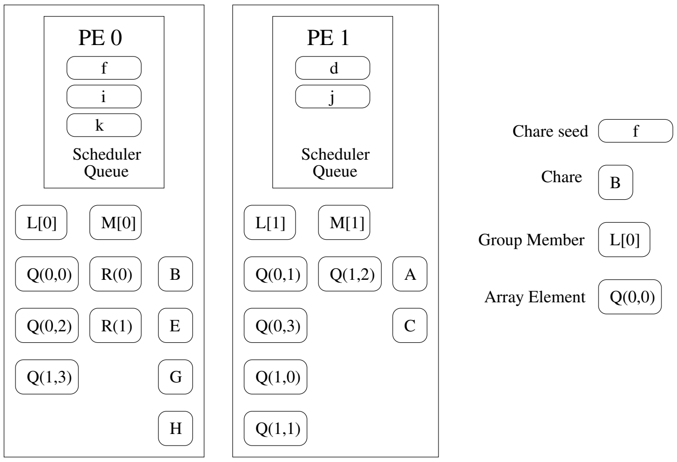

Figure 4. A typical object distribution on two processors in a Charm++ program.

Figure 4 shows various objects in a Charm++ program. Several chares have been created on the two processors, and several chare seeds are awaiting creation in the message queue. Two object groups, `L` and `M` have been created, each with one object on each processor. Finally, two object arrays, `Q` and `R`, have elements distributed across the processors.

Charm++ is built on top of the Converse run-time framework. Converse provides portable, efficient implementations of all the functions typically needed by a parallel language of library. For example, Converse provides an architecture-independent interface to most thread functions, thread scheduling, synchronization of variables, and message passing. The Charm++ environment can run on combinations of many different operating systems, including Linux, Irix, Solaris, Windows NT as well as many specialized parallel computers like Cray T3E, IBM's SP, etc.

4.2 The Emulator Interface

Writing an application for our emulator requires: 1) defining the initialization functions specifying the basic architectural features of the machine and 2) writing the application code, creating a set of handler functions that can be run on each node. Once the application is created, it is compiled and linked into the emulator library and the application is simulated on existing platforms. The same application code, sans calls to the emulator API, will be then able to compile on the actual machine.

The following initialization functions are supported by the emulator, that are relevant only for the emulation and not part of a real-machine program.

- *void BgInit(Main *)* - This user-defined function is called by the runtime system, where the execution begins. The machine is configured by calling *CreateBlueGene* in this function.
- *void CreateBlueGene(..)* - is used to specify machine configuration parameters, such as the number of nodes, the number of processors per node, etc. This function triggers the execution of various application threads on the emulated processors.

4.3 Emulator Implementation Details

The petaFLOPS machine discussed in this paper is composed of a three dimensional grid of nodes. The nodes are modeled by a Charm++ 3-D object array. The other emulator objects (Input Buffer, Scheduler Queue, threads) are implemented using a combination of Charm++ and Converse. Charm++ is used for machine initialization and sending packets from node to node. Converse is used as an interface to thread creation and scheduling. The use of Charm++/Converse ensures that the emulator can be run on multiple operating systems without changing the emulator code.

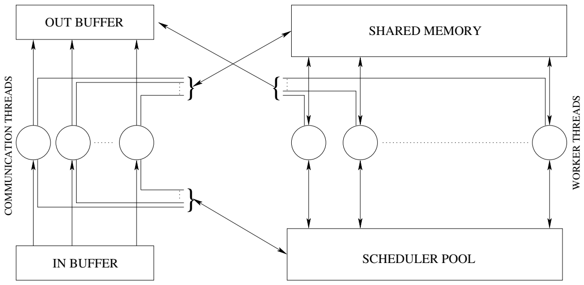

Figure 5. Functional View of a Blue Gene Node

The machine nodes are modeled in the emulator as a 3-D grid of Charm++ array objects. The Charm++ entry method *sendPacket* is the main interface for internode communication. The functional view of a node is shown in Figure 5. When a node calls sendPacket to send a message to another node located at position [x,y,z], the runtime system sends it via Charm++ to the destination node object where it is added to the *InBuffer*.

A node consists of multiple non-preemptive user-level threads divided between those primarily devoted to monitoring the InBuffer (*Communication Threads*, each emulating a P.E.) and those that are devoted to computation (*Worker Threads*). Work performed on a node is categorized into work generated by that node (*MicroTasks*) and work generated by other nodes (driven by packets arriving in the InBuffer.)

On a real machine, all the threads will run continuously, busy-waiting when no work is available for them. Such busy-waiting will be wasteful in an emulation. Instead, an efficient synchronization scheme is used, as described below.

Upon receiving a new message, the InBuffer module examines the threads in its *Communication Thread Queue*. If any communication thread is idle, the InBuffer adds the thread to the local machine's dispatch queue to be awakened. If no threads are idle, the message remains in InBuffer until an active communication thread processes it. FIFO ordering of messages ensures fair scheduling.

When a communication thread is awakened, it examines the InBuffer. If the InBuffer contains a message, the thread calls *getMessage* to extract the first message. Messages are sent with a *ThreadCategory* specifying whether a communication or a worker thread should do the work. It is efficient to execute a small piece of work directly by communication thread because of overheads involved in scheduling. For example, contributing the results of a computation in reductions should be handled by a communication thread.

If the ThreadCategory specifies a communication thread, the thread handling the message simply does the work. If the ThreadCategory specifies a worker thread, the communication thread assigns the work to a worker thread in the following manner: The communication thread first examines the threads in the *Worker Thread Queue*. If any worker thread is idle, the work is assigned to it and the worker thread is awakened. If all the worker threads are busy, the communication thread performs non-busy waiting. Worker threads behave in a simpler manner. When a worker thread resumes execution, it retrieves the message assigned to it and executes the handler function associated with the message.

Storing results of a computation accross several message executions is neccessary for any parallel processing system. We provide this functionality in the form of *Node Private Variable* (Npv). It is a data structure stored in the shared memory of each node so that all the threads can access it. Since the threads are non-preemptive, access to the Node Private Variables does not lead to any race conditions.

5 Emulator Performance Issues

To carry out emulations of realistic parallel programs meant for a petaFLOPS computer in a reasonable time requires that the *emulation slowdown* be relatively small. The emulation slowdown is defined as the ratio of time to emulate a program to the time needed on a real petaFLOPS machine. A portion of the slowdown is inevitable: emulating a system with million (GigaFLOPS) processor system on 2,000 (GigaFLOPS) processor parallel computer, a 500-fold slowdown is acceptable because we have that many fewer processors. To separate this factor, we define the notion of *emulation efficiency*: ratio of ideal emulation time for a program to the actual emulation time for it. The ideal emulation time is simply the (estimated) execution time on the petaFLOPS machine divided by the ratio of processing powers.

$$T_h = Actual\,Emulation\,Time\,on\,Host\,Machine$$

$$T_p = Execution\,Time\,on\,Emulated\,Machine$$

$$r = Ratio of\,Processing\,Power\,of\,Emulated petaflops$$
$$Machine\,to\,that\,of\,Host\,Machine$$

$$T_i = Ideal\,Emulation\,Time = T_p \times r$$

$$Emulation\,Slowdown = \frac{T_h}{T_p}$$

$$Emulation\,Efficiency = \frac{T_i}{T_h}$$

The emulation efficiency is affected by the scheduling overhead in the emulation, and the communication cost during emulation. We next describe optimizations we implemented to increase emulation efficiency.

5.1 Charm++ optimizations

Charm++ is designed for relatively coarse grained computations, compared to those needed on a petaFLOPS computer. The typical average grainsize (amount of computation per message) in Charm++ is of the order of several milliseconds or higher. Thus, if the overhead associated with a message (either between objects within a processor, or objects across processors) is even as large as a hundred microseconds, the application performance of typical Charm++ programs is not affected significantly. As a result several inefficiencies had crept into the Charm++ system that needed to be optimized for the efficiency of the emulation.

Event scheduling overhead: The raw overhead for scheduling events via the Charm++ system is typically a few microseconds (around 2-3 microseconds for short messages on Origin 2000). However, we found that the emulator required over 100 microseconds for this purpose. The difference was accounted for by dynamic allocations of several small structures, which was quite slow on the machines we used. With a few techniques for reusing messages, taking advantage of the fact that the messages were of limited (short) length, we optimized the overall performance close to the level supported by Converse, to under 3 microseconds per event.

The communication cost is an important factor in emulation efficiency. If two nodes are emulated on the same physical processor, the communication between the two nodes is modeled by local message passing. If they are on different processors, the emulation involves sending across-processor messages. The cost in two cases differs substantially, by a factor of 20 or more. As an aside, it was observed that messages that cross physical processor boundaries took much more time than expected; again because of dynamic allocation of messages. Reducing this factor by eliminating dynamic allocation brought the across-processor message overhead to around 40 microseconds per message.

The communication overhead can therefore be minimized by using a block decomposition of the node array onto the processors. (Due to the almost inevitable bisection bandwidth limitations of petaFLOPS machines, the algorithms running on them will typically involve near-neighbor communication. The block decomposition makes a large fraction of neighbor communication local to the physical processor doing the emulation). Further optimization of the communication overhead is possible by combining multiple messages going to the same physical processor into one message, using an appropriate buffering algorithm. The latter technique is currently being implemented, whereas the blocking has been implemented in the emulator.

The Charm++ system stores a map of the likely location of each object. With millions of objects, such a map will be highly space ineffcient. This map has been optimized so that it caches only the relevant objects (with a fallback mechanism for the remaining objects).

Stack-size: The stack allocated to each thread could affect the capacity of our emulation significantly. Essentially, the problem with the stack is that it requires space to be allocated based on a "high-water-mark" of usage, for the entire duration of its use. On petaFLOPS machines, one must limit the stacksize (by storing most of the data on the heap instead). We used the Charm++ capabilities to allocate short stacks (2KB) to emulate each PE.

With these optimizations, we were able to emulate relatively large configurations on parallel machines. Specifically, we did simulations on the ASCI-Red machine. On ASCI-Red machine, we could model $34 \times 34 \times 36$ (41,616) node with 200 threads per node on 96 physical processors for programs that don't use the full memory of each Blue Gene node such as the benchmarks for broadcasts and reduction described in the next section.

5.2 Emulation Benchmarks

In order to measure emulation efficiency, we needed an estimate of execution time on the emulated petaFLOPS machine. This was achieved by adding timestamps to messages and having each thread maintain its current time. (A true simulation would require more sophisticated timing mechanisms.)

Broadcast: We implemented the following two algorithms for broadcasting data to all the nodes using the emulator.

- *Line Broadcast* - In Line Broadcast, the overall pattern of transmission is a stream of messages originating on one face of the machine's cube traveling to the opposite face of the cube. The message originates in the corner of the cube of nodes, so there is some sharing of message creation and propagation in staging the initial broadcast from the corner of the cube to the face.

- *OctTree Broadcast* - In OctTree Broadcast, the message originates in the center of the cube of the nodes. The origin location partitions the cube of nodes into eight parts and recursively propagates the broadcast to each of the eight sub-cubes. Of the two methods described here, this one has the greater ratio of message creation to propagation for a theoretically faster broadcast.

The benchmarks were run on the ASCI-Red with 96 processors. It was observed that line broadcast would take 110 microseconds to complete on a full scale bluegene machine, where as the broadcast based on OctTree takes 8.9 microseconds, given our assumptions about the basic time constants: 1 nanosecond for each hop of communication and 75 nanoseconds for turning a corner, 1 microsecond for execution of handler within a thread.

It was observed that emulation of the OctTree Broadcast took 1.95 seconds, while that of the Line Broadcast took 1.99 seconds on the 96 processors of ASCI-Red. Although both emulations processed around 42000 messages, the OctTree emulation utilizes the host processors more efficiently with higher parallelism.

Reduction: We emulated a simple reduction algorithm which computes the global maximum. The algorithm selects the maximum number in an array on each node. Each node waits for data from its predecessors and then contributes the result to the next node in the reduction sequence. The reduction sequence follow the path of the Line Broadcast in reverse. Our emulator calculated that the time for the reduction algorithm would be around 95μs on emulated machine, and it took around 1.25s to run on the host machine (ASCI-red).

We implemented two small applications using the environment provided by our emulator. One is a prototype molecular dynamics application based on some of our prior work [8], which implements the core functionality of electrostatic force calculation and integration. The other application performs the Jacobi relaxation.

6 Summary and Future Work

In summary, we have shown that it is indeed feasible to do a thread-level simulation of a million-processor petaFLOPS computer on existing technology. IBM's Blue Gene architecture was used as a specific example. We described our initial scheme for modeling a massive "processor-in-memory" architecture through a parallel emulator, and have presented preliminary performance data.

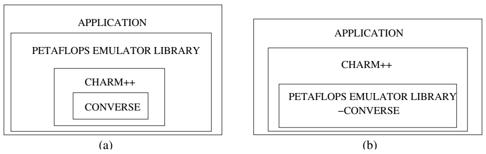

Figure 6. Proposed Software Hierarchical Layers

We have plans for several areas of research to further optimize our existing emulator and build applications using our emulator. We plan to provide a version of Charm++ as a possible programming environment for Blue Gene and other simulated machines. Implementing Charm++ on the emulator which itself is implemented in Charm++ presents name conflicts (see Figure 6a). The proposed software hierarchical layer that solves this problem is shown in Figure 6b. The emulator and programming environment API will exist at the most basic thread and message-passing level of Converse. This will enable us to implement a version of Charm++ optimized for a petaFLOPS architecture. Thus all existing programs that are currently built on Charm++ will be able to run on the emulator. Charm++ implements an automatic object-based load balancing strategy which will need to be updated for our emulator to take into account the topology of the multi-processor nodes and the communication patterns that develop in a scientific application.

A much larger undertaking is to enhance the emulator into a full fledged simulator capable of accurate performance prediction. This will require research on techniques for accurate modeling of components such as memory hierarchies and communication networks along with the contention for resources.

Based on such an emulator, one can compare several architectural variants for their performance. Also, in a million-processor machine, there are bound to be hardware failures. We plan to simulate the impact of missing processors on overall performance.

7. Acknowledgments

We thank Dr. Marc Snir, Dr. Jose Moreira, Dr. Manish Gupta and the entire Blue Gene team at IBM T.J. Watson Research Center at Yorktown Heights, NY for discussions on Blue Gene and petaFLOPS architectures. We also thank the group members of the Parallel Programming Laboratory at University of Illinois Urbana-Champaign.

References

[1] IBM announces $100 million research initiative to build world's fastest supercomputer, December 1999. http://www.research.ibm.com/news/detail/bluegene.html.

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

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

[4] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[5] Robert K. Brunner and Laxmikant V. Kalé. Handling application-induced load imbalance using parallel objects. Technical Report 99-03, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, May 1999. Submitted for publication.

[6] L. V. Kalé and Attila Gursoy. Modularity, reuse and efficiency with message-driven libraries. In *Proc. 27th Conference on Parallel Processing for Scientific Computing*, pages 738–743, February 1995.

[7] Robert K. Brunner and Laxmikant V. Kalé. Adapting to load on workstation clusters. In *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 106–112. IEEE Computer Society Press, February 1999.

[8] R. Brunner, J. Phillips, and L.V.Kalé. Scalable molecular dynamics for large biomolecular systems. In *Proceedings of SuperComputing 2000*, 2000. To be published.

[9] W. Dally et al. The j-machine : A fine grained concurrent computer. In *Information Processing 89, Proceedings of the IFIP Congress*, pages 1147–1153, August 1989.