A Parallel-Object Programming Model for PetaFLOPS Machines and Blue Gene/Cyclops

Gengbin Zheng
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
{gzheng, asingla, unger1, kale}@cs.uiuc.edu

Abstract

One approach for building the next generation of parallel computers is based on large aggregates of multiprocessor chips with support for hardware multithreading. An initial design for IBM's Blue Gene/C project exemplifies this approach. Such a machine might consist of a million processors, and is characterized by a low memory-to-processor ratio. To study alternate programming models for such a machine before it is built, we have developed an emulator that allows million-processor programs to be run on conventional parallel machines with hundreds of processors. Here we present the implementation of a parallel object model based on Charm++ as a candidate programming model. Although the "ideal" programming model for such machines is a matter of continuing research, we believe that parallel objects represent a good starting point. This paper reviews the target architecture, presents the programming model, and describes the emulator implementation. Case studies of simple applications written using the emulator are also discussed.

1. Introduction

Emerging applications such as protein folding, biological molecular dynamics, and computational quantum mechanics demand unprecedented amounts of compute power. Computer hardware designers have made incredible performance gains over the past half century; and the trend toward more powerful and cost-effective systems shows no sign of slowing.

The particular class of machines we focus on here are those composed of a large number of identical chips, each of which contains multiple processors and memory modules. For brevity, we call such machines "Massively Parallel Processors-In-Memory", or MPPIMs in the rest of the paper. Designs are currently being drafted for machines with in excess of one million processors, with petaFLOPS-class performance. The motivating machine architecture for our research is an initial design for the Blue Gene/C machine, a more advanced possible successor to the Blue Gene/L machine being developed now. Section 2 describes the Blue Gene/C architecture. Architectures like FlexRam [1], HTMT [2], Shamrock [3], RAW [4] and IRAM [5] are also relevant to this category of machines.

Although it now seems clear that such a machine can be built, and that their theoretical performance will be enormous, an important unanswered question is whether it will be cost-effective to develop software to exploit that performance efficiently. What programming model is appropriate for such a large scale machine? This paper reports on our effort to develop a model that can be used to effectively program MPPIMs. The model is based on our parallel objects system, Charm++ [6].

In order to experiment with alternate programming models, and algorithms, and to study performance issues, we are building a framework to emulate million-processor programs [7]. The initial implementation of the framework allows one to develop, debug and test programs for MPPIM machines using conventional parallel machines (such as clusters) with only hundreds of processors. Such programs will run, almost unchanged, on a real machine when one becomes available. Later stages of the framework will enable accurate time-stamping of events in order to make performance predictions for specific application-machine combinations. We describe this emulation framework in Section 3.

Section 4 explains why the parallel object model is well-suited for programming a MPPIM. The implementation of this model on the emulator presented several technical challenges, which are discussed in Section 5, along with preliminary performance data in Section 6. This paper presents a preliminary snapshot of an ongoing project. Some of the planned research is described in Section 7.

2. MPPIM Architectures

The architecture of a MPPIM is likely to differ substantially from that of today's PCs and clusters. The most likely and obvious difference is that to attain petaFLOPS performance using conventional processors will require an extremely large number—hundreds of thousands or millions—of individual processors.

Assuming the goal of a million processor system, the major challenges for building it involve power, space, and monetary budgets. It would be impractical to attempt to achieve this level of parallelism using commodity personal computers, which dissipate hundreds of watts of power, take a substantial amount of space, and cost about $1,000 per processor. Even if the peripherals unnecessary for parallel computing are removed, these numbers still remain formidable when multiplied by a million. It seems clear that a million processor cluster is not likely; and that to build a million processor machine, the total component count must be dramatically reduced.

Given chip densities of the near future, one way to reduce the chip count will be to integrate multiple processors and their memory onto a single chip.[1] Chip area, however, still limits the amount of memory per chip that can be economically used. Thus one likely feature of MPPIM machines, and an especially striking feature of IBM's Blue Gene/C, is an extremely small amount of memory per processor.

Since the million processors of a MPPIM have to be packed into 3D space, physical cabling limitations mean the machine's topology is likely to be 3D mesh or torus. This means the cross-section bandwidth is likely to be relatively small, and the number of hops to cross the machine relatively large. This, in turn, implies that machine topology may again become an important factor in the design of parallel algorithms and implementations.

We consider IBM's Blue Gene/C an example of the class of MPPIMs. It aims to reduce costs and the component count by fabricating several processors (25 in one design), along with their memory, onto a single chip, or *node*. It also attempts to improve processor utilization by including eight hardware threads on each processor. Each node, then, has 200 hardware threads, which share only about 16 megabytes of memory. The threads in a node all access the same memory, as in a conventional SMP, but a memory-side caching architecture eliminates the need for a cache coherence protocol. Communication between nodes is via explicit message passing, connected in a 34 x 34 x 36 grid of nodes. The machine works out to a petaflop of raw performance, a terabyte per second of cross-section bandwidth, and only half a terabyte of total memory.

These specifics are particular to IBM's Blue Gene/C machine, but the overall architecture—many SMP single chip nodes connected via message passing on a packet-switched 3D mesh—is likely to be the most cost-effective way to build any very large-scale machine.

3. Blue Gene Emulator

One important factor which makes emulation of a MPPIM with a smaller machine feasible is the low memory-to-processor ratio on the emulated machine. Still, Blue Gene, the MPPIM under consideration in this research, is likely to have about half a terabyte of total memory, which makes it impractical to emulate using a single processor. However, emulating a MPPIM running an application which uses the full machine will require "just" 1000 processors of a traditional parallel machine with 512MB per processor.

A parallel emulation poses another potential problem: messages in the emulator may be delivered in a different order than messages sent in the real machine. Therefore, the application must be written so as to handle out of order message delivery. Fortunately, this is not a significant obstacle, as many parallel programming paradigms and their runtime systems already handle this behavior.

We chose to support a low-level, but fairly general, API in the emulator [7], as described below. The emulator is implemented using Converse [8], and runs on a variety of parallel machines including clusters and supercomputers. The emulator is available via the web at http://charm.cs.uiuc.edu.

A low-level API allows one to build higher-level 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) that share memory within a single node. Each thread has its own stack and a work queue. Access to a reliable communication layer is provided via calls to send short (limited-length) messages to other nodes, along with an index of the handler to be invoked on the destination. For Blue Gene, message length is limited to around 100 bytes.

[1] Recent advances in semiconductor manufacture now make it possible to integrate logic and DRAM on a single chip.

Figure 1 describes the functional view of a Blue Gene/C node in the Emulator.

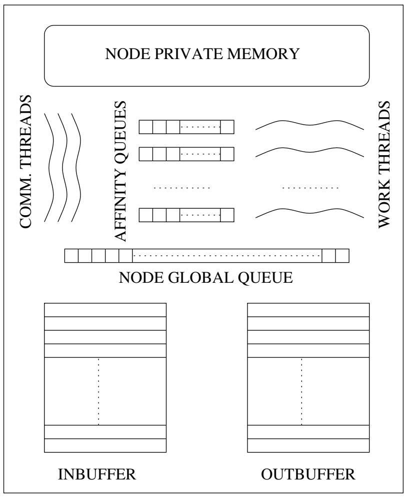

Figure 1. Functional view of a Blue Gene node in the Emulator

Within a node, we divide threads into worker threads and communication threads. Communication threads check for incoming messages from the network and put the messages in either a worker's queue or a node global queue. Worker threads repeatedly retrieve messages from the queues and execute the handler functions associated with the messages. In order to make use of locality of data for performance improvement, *affinity queues* are created for each worker thread. When a message for a specific thread ID arrives, the communication thread will schedule it to the affinity queue of that particular worker thread. This is done to ensure that messages are executed in the same place where the object data needed by the handler function is located. A worker thread will process work from its own queue before checking the global queue.

The 3-D grid of Blue Gene nodes are mapped to the physical machines in a round-robin fashion in the preliminary implementation. *BgSendPacket()* is the main interface for internode communication. Messages can be sent with a *WorkType* parameter specifying whether a communication thread or a worker thread should do the work. It can be more efficient to execute a small piece of work directly in the communication thread to avoid the overheads involved in scheduling.

Code in a worker thread may create work for another thread on the same node by firing a *micro-task*. The message describing a micro-task may contain pointers to data on the same node.

Below is a summary of the current emulator API.

- *void BgNodeStart()* - is called by the runtime system to initialize each node. Here, application handlers are registered, and the computation is started by creating tasks for the specified nodes.
- *int BgRegisterHandler(BgHandlerFn h)* - is invoked to register a handler function with each node and return a globally unique identifier associated with it.
- *void BgAddMessage(int threadID, int handlerID, int nBytes, char *msg)* - is called to create a task on the local Blue Gene node.
- *void BgSendPacket(int x, int y, int z, int threadID, int handlerID, WorkType type, int nBytes, char *data)* - is used to send a message to a node at location [x,y,z] in the node grid. ThreadID can be used to direct a task to a thread, otherwise any thread in the node can handle the message.
- *Utility functions* - In addition, the emulator supports several utility functions that allow access to timers, the identity of the node and the processor on which the invoking thread is running, and other housekeeping data.

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

4. Why parallel objects for MPPIM

Conventional models such as message passing, shared memory and data parallel programming need a lot of programmer effort to efficiently utilize a MPPIM due to problems with load balance, locality, and parallelism. Charm++, as a parallel object programming model, has several advantages compared to these conventional models.

Charm++'s parallel objects provide a degree of freedom to the run-time system that is very helpful for programming MPPIMs. In this model, the programmer specifies the decomposition of the problem only in terms of interacting objects, and the run-time system handles mapping (and remapping) these objects to processors. Thus, in the programmer's view, object A invokes a method in object B, but the programmer doesn't know or care which processor object B resides on. Several different objects are often mapped to a single processor, and scheduling within a processor is accomplished via a message-driven non-preemptive scheduler. The scheduler selects a method invocation (also called a message) from the queue, identifies the object it is intended for, and invokes the method. When the method invocation returns, the scheduler picks the next message.

Objects may even migrate from processor to processor at runtime, usually under the control of a system-supplied load balancer. Message forwarding may be required after a migration, but the system uses a routing scheme that asymptotically requires only 1 hop (i.e. no indirection) for any kind of repeated communication [9].

The flexibility provided by this object model is a key to the high performance attained by Charm++ based applications which had been hard to parallelize otherwise, as shown with NAMD [10]. Charm++'s automatic load balancing, based on measurement of computation loads and communication patterns among objects, can lead to significant improvements in performance with very little additional effort by the programmer. This is especially true for adaptive applications, which change their computational characteristics with time.

For a MPPIM machine, locality is extremely important because of the potentially high cost of accessing remote data (on a large diameter topology). Thus, object-based decomposition is particularly attractive because it models locality well. Objects encapsulate state, and Charm++-style objects are allowed to directly access only their own local memory. They may make direct calls to other objects guaranteed to be on the same processor, and access readonly data that is accessible to all objects. However, access to any other data (e.g. data in other objects) is only possible via asynchronous method invocation. This enforces locality in a clear manner, and the programmer is aware of the cost of accessing remote data.

Load balancing on MPPIMs is more difficult than load balancing conventional machines. For example, while mapping objects to processors on machines with hundreds of processors, it is often possible to ignore the number of hops traveled by messages. This is because the latency is almost independent of the number of hops with modern techniques such as wormhole routing, and the impact on bandwidth utilization is limited. For a machine with 40,000 nodes, where cross-processor messages require many hops, the bandwidth used in the intermediate links becomes a significant concern. In Charm++, load balancing can be handled by the run-time system, without changing the programmer's view of their applications. Thus only the load balancing run-time needs to be changed for a MPPIM machine, which can simultaneously improve performance and reduce programmer effort.

One of the questions that arises while programming MPPIMs is how to generate the large amount of parallelism needed to occupy the millions of processors. The object model allows a solution: since the users decompose the problem into objects, and since they decide the granularity of objects, it is easy to generate parallelism. The object model imposes no arbitrary restrictions on the decomposition, such as the requirement that the problem be decomposed into as many pieces as processors often encountered in other models. For an example of how to generate parallelism using objects, see the LeanMD program discussed in Section 6.

5. Blue Gene Charm++

We implemented Charm++ on the Blue Gene emulator to allow us to more easily study the performance and scalability of a MPPIM machine. In this section, we describe some design issues involved in the implementation of Blue Gene Charm++, along with some optimizations to improve the efficiency of Charm++ and the Blue Gene emulator.

5.1. Design Issues

In Charm++, the programmer doesn't concern themselves with which processor an object resides on. In the implementation, however, objects have to be mapped to processors by the Charm++ run-time system (RTS). Furthermore, Charm++ semantics requires that two methods of an object never execute concurrently (method atomicity). In conventional implementations, Charm++ groups together the objects and threads that reside on the same processor. On each processor, there is one non-preemptive scheduler responsible for scheduling messages and doing method invocations associated with messages, which ensures atomicity. Even on machines with SMP nodes, the Charm++ RTS divides the objects by processor and schedules each processor separately, rather than using one shared scheduler. This guarantees method atomicity. On Blue Gene, we considered two alternative strategies to guarantee atomicity.

1. Treat an entire Blue Gene node (with many processors and threads) as a single Charm++ processor. (So, on the Blue Gene design mentioned earlier, there will be about 40,000 Charm++ "processors"). When a message is sent to a Charm++ processor, the emulator communication threads can schedule the message to any worker thread. This requires using locks on objects to ensure that two method invocations do not execute concurrently. However, no such locking is necessary during emulation, since all the threads on a MPPIM node are mapped to a single emulator processor, where is no preemption.

2. Treat each worker thread of MPPIM as a Charm++ processor (so there are several million Charm++ processors in the above example). Chare objects are anchored to individual threads. When a message arrives, it will be scheduled into the affinity queue of the worker thread specified by the message. This approach requires explicit load balancing among threads of a node.

In this paper, the work we present is based on the first scheme. In addition to the advantages mentioned above, it also needs a smaller amount of memory for per-processor data structures maintained by the RTS.

Charm++ is implemented using Converse as shown in Figure 2. The Converse run-time framework provides portable, efficient implementations of all the functions typically needed by parallel applications. For example, it provides a common interface to the machine dependent implementations of thread creation and message passing.

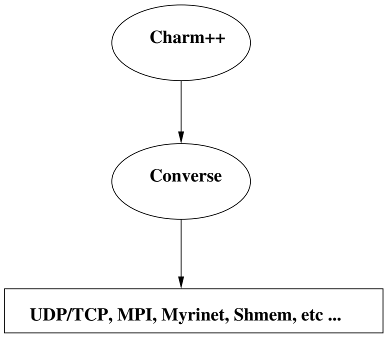

Figure 2. Original Charm++ System hierarchy

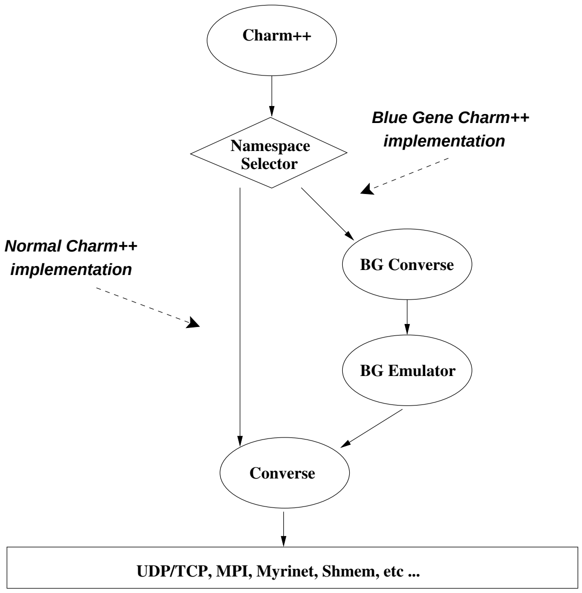

Figure 3. Layered Implementation of Blue Gene Charm++

For Blue Gene Charm++, we needed a Converse layer that uses the Blue Gene emulator API, which should provide the same functionality of the original Converse. However, the emulator that this Blue Gene Converse is built on, itself is implemented upon the original Converse. That is, the same Converse calls are used in two contexts: sometimes to represent the real machine– Converse; and sometimes to represent the virtual emulated machine–Blue Gene Converse.

While the two uses of the Converse share the same interface, the implementations are completely different. For example, Processor Private Variables in the Converse layer are private to each real physical processor; while in Blue Gene Converse, Processor Private Variables are private to each Blue Gene node. This causes significant name conflict problems.

To solve this problem, we separated a component layer from Converse that consists of all Converse calls used by Charm++ runtime. Fortunately, Charm++ only needs a small subset of the entire Converse API. We then implmented this layer on the Blue Gene emulator. A similar implementation, consisting of simple wrappers, was created for normal Converse as well. Each of these were encapsulated in different *name spaces*. This layered implementation is showed in Figure 3. Using C++ namespaces, the Charm++ runtime can now easily switch between the Blue Gene Converse and the normal Converse. This greatly simplifies the implementation while allowing two versions of Charm++ to coexist in one system. Here is an example of what a routine in the Blue Gene Converse version looks like:

```
namespace BGConverse {
 ...
 void
 CmiSyncSendAndFree(int pe, int numBytes,
                              char *msg)
 {
  int x,y,z;
  // find out the coordinates of
  // blue gene node.
  BgGetXYZ(pe, &x, &y, &z);
  BgSendPacket(x, y, z, ANYTHREAD,
                  CmiHandler(msg),
       LARGE_WORK, numBytes, msg);
 }
}
```

With these design decisions, an implementation of Blue Gene Charm++ was created that shares the code base with the rest of the Charm++ RTS, which is now available on the web (`http://charm.cs.uiuc.edu`) along with our normal Charm++ distribution.

5.2. Charm++ and Emulator optimizations

Blue Gene Charm++ and emulator are designed for simulation of millions of processors on limited traditional machines. The huge number of Blue Gene processors, reflected in emulator as user-level threads, has a great impact on the emulator and Charm++ implementation. As a result, the thread scheduling, message passing and memory efficiency in emulator and Charm++ implementation needed to be optimized for efficient simulation.

Emulator efficiency: In [7], we demonstrated that the emulator is able to model 34 x 34 x 36 grid with 200 threads per node on 96 physical processors of ASCI-Red after several optimizations. (Note that this entailed creating approximately 80,000 user-level threads per ASCI-red processor) However, with Charm++ – another layer on top of the emulator, the efficiency of the emulator becomes more critical. The original implementation of the Blue Gene emulator, which was developed on top of Charm++, was less efficient and made it difficult to emulate Charm++ programs.[2] In an emulator built on a higher level system, the overhead of thread scheduling and communication are relatively higher. To reduce these overheads, we completely rewrote the Blue Gene emulator at the Converse level, while preserving the emulator API. The new Converse-based Blue Gene emulator is closer to the machine level and consequently substantial overhead was eliminated. Specifically, the emulator's one-way pingpong per message over ethernet on Linux clusters has improved from 134us to 92us, which is close to the 83us pingpong time of bare Converse.

Charm++ run-time overhead: In Blue Gene Charm++, the memory allocated by Charm++ can greatly affect the capacity of our emulation. However, the original Charm++ implementation incurred high memory cost with millions of processors. It stored various internal data structures (tables) for Chares, Group and Arrays on each Charm++ node. In many cases, to ensure table lookup speed, these tables are allocated that take space proportional to number of processors, which caused significant space inefficiency when simulating millions of processors. Unlike in original Charm++,in Blue Gene Charm++ optimizations for space efficiency are of higher priority than optimizations for speed. Therefore, those data structures that take space proportional to the number of processors were altered to be allocated dynamically only if needed.

Another optimization to reduce the memory usage is table sharing: tables that store global object information, such as read-only data and object location caches, don't have to be duplicated on every Blue Gene node. Only one copy per *emulator* processor is needed.

6. Case Studies

With an implementation of the Charm++ programming model on the Blue Gene emulator, nearly all existing Charm++ applications are able to run on the emulator. This software reuse allows quick studies of Blue Gene performance issues on conventional supercomputers or clusters before the machine is available.

[2] A system built on Charm++ will have more serious name conflicts when trying to emulate Charm++.

In this section, a simple but typical Charm++ program is studied for the emulator efficiency issues; then, we will describe a realistic application - LeanMD, which is a prototype application for a general purpose molecular dynamics simulations. The results presented here are preliminary.

Blue Gene Charm++ efficiency: How efficient is Emulator based Charm++ considering the fact that it is implemented via multiple layers(Fig. 3)? To demonstrate the efficiency of the Blue Gene Charm++, we compared the performance of Blue Gene Charm++ with the original Charm++.

We chose to run a small Charm++ program that calculates the value of π. In this program, node 0 creates 1000 Chares distributed to all Blue Gene nodes, each Chare does its own work of sampling and returns its result to node 0. The performance data on a Linux cluster is shown in the table 1. We can see that the overhead of emulator is actually pretty small and Blue Gene Charm++ is efficient and comparable to the original Charm++.

processors	1	2	4	8	16
BG Charm	137.11	70.33	34.43	17.22	8.69
Charm	137.74	69.53	34.58	17.30	8.71

Table 1. Comparison of BG Charm++ and original Charm++ of PI. All timings are in seconds.

LeanMD: With these encouraging test results, we were ready to run full-fledged Charm++ applications on Blue Gene emulator. As an example, we chose LeanMD to study its performance.

The need for a parallel programming paradigm like Charm++ for the Blue Gene/C architecture becomes apparent when considering its primary application: protein dynamics studies via molecular dynamics. Current state-of-the-art parallel molecular dynamics applications do not scale well to thousands of processors, with a few exceptions such as NAMD [10]. However, scaling to millions of processors will clearly require new parallelization strategy beyond even that of NAMD (a molecular dynamics application developed in Charm++). The problem must be broken up in a more fine-grained manner to effectively distribute work across the millions of processors. An object-oriented paradigm for parallelism like Charm++ can directly address the problem of how to break-up the problem into finer objects. In addition Charm++ provides proven load balancing capability.

In NAMD, the atoms in the simulation are divided spatially into cells roughly the size of the cutoff distance. Local interactions are calculated each timestep between only the nearest neighbor cells ("one-away" interactions), as illustrated in Figure 4. This ensures that all atoms within the cut-

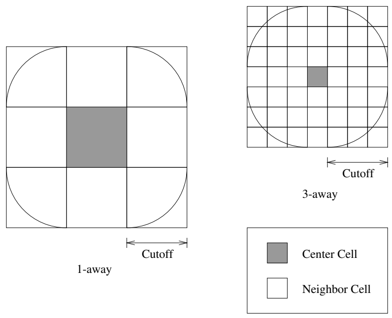

Figure 4. 1-away vs. 3-away cell cutoff distance.

off radius are calculated. However, this strategy produces a division that is coarsely grained for Blue Gene/C. For example, with a cutoff radius of 15 Å, a 150 x 150 x 150 Å simulation space would give only 1,000 cells and 13,000[3] cell-to-cell interactions to calculate. Considering that the Blue Gene/C machine is approximately 40,000 nodes, the division would leave nodes idle even if interactions were delegated to a single node.

To address the issue of creating finer-grained parallelism for cutoff interactions, an experimental program called LeanMD is being developed. In LeanMD, the "one-away" strategy is replaced with a "k-away" strategy. Instead of one cell representing the cutoff distance, in LeanMD three cells would span the cutoff distance as shown in Figure 4. Therefore, in order to do the cutoff calculation, a cell must compute its interactions with every cell that is "three-away" in this scenario. Given the simulation example above, a three-away strategy would produce 27,000 cells and more than 4 million cell-to-cell interactions, a number of objects that is easily distributed across the 40,000 nodes of the Blue Gene/C.

Charm++ facilitates the object tracking and distribution via its array formulation. The physical cell grid is modeled using Charm++'s built-in three-dimensional array indexing, and the cell-to-cell interactions are addressed by a six-dimensional array. The array formulation allows the six-dimensional array to be sparse, and elements are distributed across the machine via the Charm++ array manager. The load balancer will further distribute elements in both arrays during run-time.

The LeanMD project is currently under development. Here we present the preliminary performance data on the Blue Gene Charm++ and emulator. This test used brH simulation benchmark, which consists of 3762 atoms. The simulation is 3-away with cutoff of 12 Å, the cell size thus is 4x4x4 and the simulation space is 11x14x11 cells. In the simulation, the number of cell-to-cell interactions is 144914, and we emulated Blue Gene/C nodes of size 10x10x10 with 200 threads each node(200,000 threads total). Table 2 shows the emulation time per step of LeanMD on various number of processors, and figure 5 shows the efficiency curve run on a Linux cluster. It shows that increasing the processors used by the emulator, we achieved superlinear speedup for the emulation. (The superlinearity is due to a smaller memory footprint per emulator processor as the number of processors is increased. The total amount of memory needed for emulation can be estimated to include at least 400MB for stack space for the 200K threads, in addition to relatively smaller amounts for RTS data structures and application data)

[3] 1,000*27/2, considering that cell-to-cell forces are symmetric.

num of pes	4	8	16	32	64
time/step(sec)	13.0	6.51	3.06	1.53	0.776

Table 2. Time per step in seconds of LeanMD

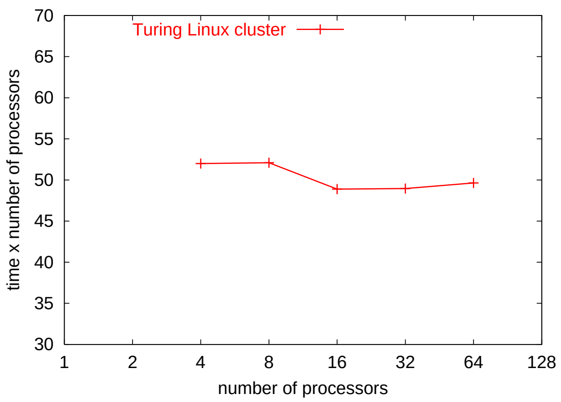

Figure 5. Cost curve for LeanMD on Blue Gene Charm++. Linear speedup would result in a horizontal line.

7. Summary

We have demonstrated a programming model and an initial emulator for MPPIMs, a class of extremely high-performance computers that includes IBM's Blue Gene machine. In the future, we are planning to modify the emulator to achieve at least first-order accurate timing data, thus transforming it into a simulator. Modeling of the network, including contention, will also be carried out. Further, in collaboration with Prof. S. Adve, we plan to incorporate accurate architectural models into the simulation. A simultaneous multithreaded (SMT) processor model for the RSIM simulator [11] has already been developed.

To study the application behavior on such large scale machines, we are creating an experimental high-performance computational molecular dynamics program, LeanMD. We are also working on developing large adaptive Finite-Element based structural simulation application using the emulator [12] in collaboration with Prof. P. Geubelle.

Implementation strategies for Charm++ itself will be further optimized. The issue of whether objects should be anchored to individual threads or whole nodes (See Section5.1) will be further explored. Load balancing strategies probably represent the most significant challenge, as well as opportunity, for our model. A new class of strategies are being developed that work in a distributed manner (without serial bottlenecks during load balancing), and can take into account interconnection topology as well as multi-level locality (i.e. objects co-located within a thread, processor, or node). Higher level languages and compilation issues are being studied in collaboration with Prof. D. Padua.

8. Acknowledgements

This research was funded in part by an NSF grant under the Next Generation Software (NGS) program, via NSF EIA 0103645, and by NIH via PHS 5 P41 RR05969-04. We also thank our group members of the Parallel Programming Laboratory at University of Illinois Urbana-Champaign, especially Orion Lawlor. We are grateful to the Blue Gene team at IBM, including Dr. Manish Gupta and Dr. José Moreira , as well as Dr. Marc Snir (now at Univ. of Illinois) for useful discussions earlier during the implementation of the emulator.

References

[1] Yi Kang, Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenzhou Ge, Vinh Lam, Pratap Pattnaik, and Josep Torrellas. Flexram: Toward an advanced intelligent memory system. In *International Conference on Computer Design (ICCD)*, 1999.

[2] G. Gao, K. Likharev, P. Messina, and T. Sterling. Hybrid technology multithreaded architecture. In *6th Symposium on the Frontiers of Massively Parallel Computing*, pages 98–105, 1996.

[3] P. Kogge, S. Bass, J. Brockman, D. Chen, and E. Sha. Point Designs for 100 TF Computers Using PIM Technologies. In *Proceedings of the 1996 Frontiers of Massively Parallel Computation Symposium*, 1996.

[4] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it All to Software: Raw Machines. *1997 IEEE Computer*, pages 86–93, September 1997.

[5] D. Patterson, T. Anderson, N Cardwell, R Fromm, K Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. Intelligent RAM(IRAM): chips that remember and compute. *1997 IEEE International Solids-State Circuits Conference. Digest of Technical Papers*, pages 224–225, February 1997.

[6] 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.

[7] Neelam Saboo, Arun Kumar Singla, Joshua Mostkoff Unger, and L. V. Kalé. Emulating petaflops machines and blue gene. In *Workshop on Massively Parallel Processing (IPDPS'01)*, San Francisco, CA, April 2001.

[8] 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.

[9] O. Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. In *Proceedings of ACM 2001 Java Grande/ISCOPE Conference*, pages 21–29, Stanford, CA, Jun 2001.

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

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

[12] P. H. Geubelle and W. G. Knauss. Crack propagation at and near bimaterial interfaces : linear analysis. *ASME J. Appl. Mech.*, 61:560–566, 1994.

[13] 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.

[14] H.P. Zima and T.L. Sterling. A Programming and Execution Model for DRAM Processor-In-Memory Arrays. In *Proc. First International AURORA Conference (IAC2000)*, January 2000.