ACHIEVING HIGH PERFORMANCE ON EXTREMELY LARGE PARALLEL MACHINES: PERFORMANCE PREDICTION AND LOAD BALANCING

BY

GENGBIN ZHENG

B.S., Beijing University, China, 1995
M.S., Beijing University, China, 1998

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

Abstract

Parallel machines with an extremely large number of processors (at least tens of thousands processors) are now in operation. For example, the IBM BlueGene/L machine with 128K processors is currently being deployed. It is going to be a significant challenge for application developers to write parallel programs in order to exploit the enormous compute power available and manually scale their applications on such machines. Solving these problems involves finding suitable parallel programming models for such machines and addressing issues like load imbalance. In this thesis, we explore Charm++ programming model and its migratable objects for programming such machines and dynamic load balancing techniques to help parallel applications to easily scale on a large number of processors. We also present a parallel simulator that is capable of predicting parallel performance to help analysis and tuning of the parallel performance and facilitate the development of new load balancing techniques, even before such machines are built.

We evaluate the idea of virtualization and its usefulness in helping a programmer to write applications with high degree of parallelism. We demonstrate it by developing several mini-applications with million-way parallelism. We show that Charm++ and AMPI (an extension to MPI) with migratable objects and support for load balancing are suitable programming model for programming very large machines.

It is important to understand the performance of parallel applications on very large parallel machines. This thesis explores Parallel Discrete Event Simulation (PDES) techniques with an optimistic synchronization protocol to simulate parallel applications running on a very large number of processors. We optimize the synchronization protocol by exploiting the inherent determinacy that is normally found in parallel applications to reduce the synchronization overhead significantly.

Load balance problem presents significant challenges to applications to achieve scalability on very large machines. We study load balancing techniques and develop a spectrum of load balancing strategies based on studies of the characteristics of applications. These load balancing strategies are motivated by several real-world applications such as LeanMD, NAMD (both are classical molecular dynamics applications) and

Fractography3D (a dynamic 3D crack propagation simulation program). We optimize our load balancing strategies in multiple dimensions of criteria such as load balancing for improving communication locality, sub-step load balancing, and computation phase-aware load balancing.

We further study the performance of existing load balancing strategies in the context of very large parallel machines using the parallel simulator we developed. We demonstrate the weaknesses of the centralized and fully distributed load balancing schemes via large scale simulation, and design a new scalable hierarchial load balancing scheme suitable for such large machines. This hierarchial load balancing scheme builds load data from instrumenting an application at run-time on both computation and communication pattern in a fully automatic way. The hierarchial load balancing takes application communication pattern into account explicit in decision making. It also incorporates an explicit memory cost control function to make it easy to adapt to extremely large number of processors with small memory footprint.

To my wife Lixia Shi and our newborn son Chenyuan

Acknowledgements

I would like to thank my thesis advisor, Prof. Kale, for his guidance, advice, motivation and continued support throughout my academic career at University of Illinois, without which this thesis would not have been possible.

I would like to thank the several past and current PPL members, Milind, Orion, Terry, Sameer, Eric, Chao ... It was a lot of fun working with you guys.

Many thanks to the various members of the TCBG group. Robert for helping me with load balancing framework in my thesis. It was a great pleasure and experience working with Jim.

I would also like to thank my wife Lixia, who has always been my source of inspiration. I'll never forget the day she gave birth to my son ChengYuan, who brought endless joyfulness to my family.

Finally, I would like to thank my parents for all their love, encouragement and inspiration that was needed to accomplish this task.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Emerging applications such as computational biology, computational cosmology, engineering design, earthquake modeling and weather forecasting demand an unprecedented amount of compute power. Computer hardware designs have achieved incredible performance gains over the past half century, and the trend toward more powerful and cost-effective systems continues. As an example, the US Department of Energy's Advanced Simulation and Computing [3] (ASC) program has delivered a series of increasingly powerful platforms to the national laboratories, including ASC Red Storm (10K processors and 40 TFlops), Purple (12K processors and 100 TFlops), and BlueGene/L [5] (180/360 TFlops) by IBM has 64K dual-processor nodes. An earlier ambitious design from IBM, code-named Cyclops (Blue Gene/C), had over one million floating point units, fed by 8 million instruction streams supported by individual thread units, targeting 1 petaflops of peak performance. IBM has also revealed an ambitious program to further expand the horizons of supercomputing with the goal of creating a system called Blue Gene/P that can achieve one petaflop of peak performance.

1.1 Petaflops-class Machine Architectures

In this thesis, we focus on massively parallel machines which are capable of delivering petaflop level performance. There are two trends in the approach to building such big parallel machines. One trend is to build such machines with conventional powerful CPU chips and interconnects. The major challenges for this approach 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 each, take a substantial amount of space, and cost hundreds of dollars per processor. Another trend is to use new, low power consumption, multi-core chips with modest performance and integrated communication support. To attain petaflop performance with this approach will require an extremely large number — hundreds of thousands or even millions — of individual processors and they are likely to have relatively low memory-to-processor ratio. This will have an impact on performance and on the way the machines are programmed.

The communication hardware of peta-scale parallel machines may also differ from that of today's parallel machines. Since the large number of processors of a peta-scale machine have to be packed into 3D space, physical cabling limitations mean the machine's topology is likely to be a 3D mesh or torus. This means the cross-section bandwidth is likely to be relatively small, and the number of hops to traverse the machine relatively large. This, in turn, implies that machine topology may become an important factor in the design of parallel algorithms and their implementation, as in the academic research in early years of parallel computing [43].

IBM's Blue Gene/C [76] is an example of the class of peta-scale parallel machines. It aimed 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* (so called system-on-chip technology). It also attempted 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. All threads in a node access the same memory, as in a conventional SMP. Communication between nodes is via explicit message passing, with nodes arranged in a 34 x 34 x 36 grid. The machine is designed with a petaflop of raw performance, a terabyte per second of cross-section bandwidth, and only half a terabyte of total memory. Although this design was not chosen for full-scale implementation, a similar design is being investigated within IBM on a smaller scale.

Blue Gene/L [5], another design from IBM that was implemented, is constructed based on a similar, custom system-on-chip compute node. Each compute node is a chip with 2 cores (700MHz PowerPC440), with 256MB of main memory, which is still small compared with conventional machines. The 64,000 compute nodes are organized in a 64 x 32 x 32 three-dimensional torus, packaged from compute cards, node boards, cabinets to the whole system as illustrated in Figure 1.1 [1]. The system can deliver 360 teraflops of peak performance.

These specifics are particular to IBM's Blue Gene machines, but the overall architecture—many multi-

[1] taken from IBM Blue Gene website: http://www.research.ibm.com/bluegene/

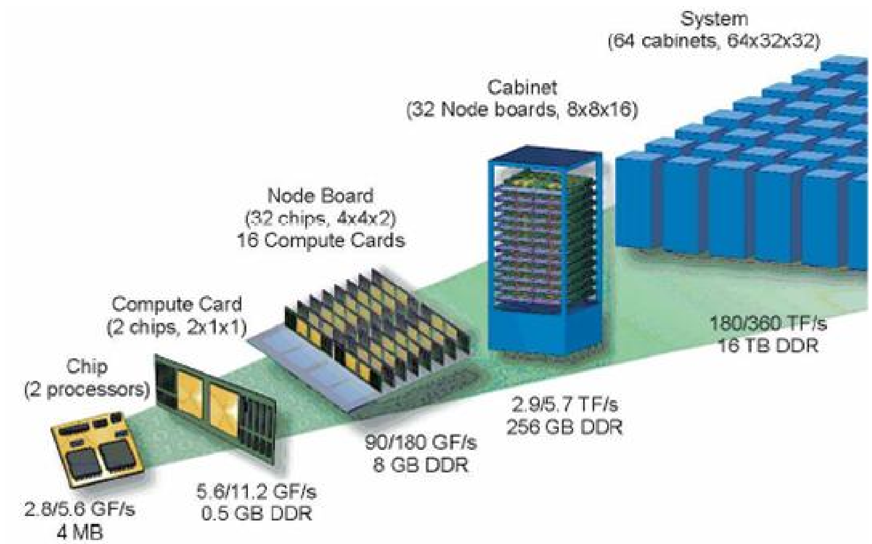


Figure 1.1: Blue Gene/L architecture (taken from IBM Blue Gene website)

processor 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.

1.2 Software Challenges

The development of parallel applications for petaflop machines is a significant challenge. It may require qualitative changes to the way we write parallel programs in order to exploit the enormous compute power, as well as changes to the way we understand and analyze the performance.

- It is very challenging to write scalable peta-scale applications using conventional programming models such as MPI, shared memory and data parallel programming languages because of the massive parallelism involved in these applications. Many scientific applications requires tightly coupled physical components, which may lead to tremendous amount of fine-grained communication among them and make the applications difficult to scale. The conventional programming models do not focus on an intelligent run-time on how to fully utilize a peta-scale parallel system.

- Molecular dynamics simulations are one of the target applications for massively parallel machines such as Blue Gene/L. These applications have very fine-grain parallelism because each time-step only involves a few seconds of sequential computation time. In order to achieve scalability on such machines, each time step needs to be parallelized into sub-millisecond of computation and communication. This implies that no software component in the parallel program can exceed this amount of time on each processor in order to achieve the desired level of performance. This again requires an intelligent run-time for resource management such as load balancing.

- Multiple interacting factors could result in performance degradation for applications on peta-scale machines. These factors include inappropriate problem decomposition, communication latencies, operating system interference, critical paths and load imbalance. Although similar performance problems apply on existing small scaled parallel machines, they tend to be magnified on peta-scale machines. Among the degradation factors, load imbalance problems are particularly significant and are quite challenging to solve given the scale of the machines. In fact, many other performance problems such as communication latencies and OS interference can turn into load imbalance problems, making load imbalance the main symptom of all performance problems. Expecting the application programmers to deal with these issues manually will be unrealistic and unproductive. Automated support via run-time system is therefore essential.

- It is especially difficult to understand how an application will perform on a peta-scale machine when the target machine is not even operational. Parallel performance is hard to model without actually running a program. Even when a machine is operational, large-processor runs are not available as often as needed. In this scenario, a parallel performance simulator is desirable and necessary as a tool for studying application performance issues. Developing such a large scale parallel performance simulator, however, is quite challenging due to the scale of the target machines simulated.

1.3 Approach

In this thesis, we propose, explore, and substantiate the following hypothesis:

> **employing migratable objects (processor virtualization) for programming peta-scale machines supported by parallel emulation for algorithm validation, and parallel simulation for performance prediction, and using new kinds of load balancing strategies**

will substantially address many of the challenges for programming petaflops class machines (described in Section 1.2).

We explore the Charm++ parallel object programming language with a data-driven execution paradigm as an alternative programming model on peta-scale parallel machines. Adaptive MPI, an MPI implementation on Charm++, enables us to apply our approach to the popular message-passing paradigm, retaining the advantage of processor virtualization.

To study how an application performs on a peta-scale machine, we adopted the simulation-based approach for performance prediction of parallel applications. We explore an optimistic synchronization approach with run-time optimization to provide a simulation based framework for studying parallel performance issues on very large parallel machines. We demonstrate that by taking advantage of user-level threads, and aggressive optimizations that reduce PDES synchronization overhead using a method that combines both language and run-time, we are able to predict parallel performance of applications on very large parallel machines with tens of thousands of processors.

With the infrastructure of the parallel simulator, we study the parallel performance of several important applications and explore dynamic load balancing techniques to improve the scalability of parallel applications in general. We develop a spectrum of sophisticated centralized load balancing strategies and achieve excellent performance for applications on modestly large parallel machines with up to several thousand processors. We further explore a category of new load balancing schemes suitable for very large scale parallel machines.

1.4 Contributions

This thesis aims at developing techniques and methods to facilitate the development of peta-scale applications. The main contributions of the dissertation include the following:

- Processor virtualization is a useful technique which makes it feasible for a programmer to write applications with high degree of parallelism that is often needed for programming peta-scale machines. We demonstrate it with several mini-applications with million-way parallelism. We show that Charm++ and AMPI (an extension to MPI), which embody the idea of processor virtualization, are suitable programming model for programming peta-scale machines.

- *A parallel simulator:* We have developed a parallel simulator that is capable of predicting parallel performance to help analysis and tuning of the parallel performance. The parallel simulator adopts parallel discrete event simulation techniques with an optimistic synchronization protocol. Such a simulator involves very fine-grain messages leading to prohibitive communication and simulation overhead. With the idea of extensively exploiting the *inherent determinacy* in parallel applications, we are able to reduce the synchronization overhead and improve the scalability of the simulator. We demonstrate the scalability of the parallel simulator by case studies of simulating a parallel molecular dynamics simulation program on a 64K processor machine.

- An enhanced load balancing framework to address the load-balancing needs of many different types of applications. It provides automatic dynamic load balancing with little user intervention. The framework performs automatic measurement of object computation times and communication patterns including collective communication, without adding undue overhead. We optimized load balancing strategies in multiple dimensions of criteria such as load balancing for improving communication locality, sub-step load balancing, and computation phase-aware load balancing.

- We demonstrate the value of these techniques by applying our load balancing framework to several real-world applications, resulting in improved performance even for applications that were difficult to balance the load. NAMD, a production quality parallel molecular dynamics program written in CHARM++, achieved the highest reported speedup on LeMieux with 3000 processors with joint work with other group members, which won the Gordon Bell award at SC2002 for the outstanding performance.

- A load balancing strategy for very large parallel machines. This thesis presents a parallel adaptive hierarchical load balancing strategy that works effectively on very large parallel machines. Having application developers provide load information is unrealistic for such large scale applications. This hierarchial load balancing strategy builds load data from instrumenting an application at run-time on both computation and communication pattern in a fully automatic way. It also incorporates a cost function to explicit control the memory overhead of load balancing to make it easy to adapt to extremely large number of processors with even small memory footprint. Our simulation of the load balancing algorithm on a machine of 65,536 processors shows that the algorithm achieves good load balance with optimization of application communication taking into account. The results also demonstrate that the parallel load balancing algorithm runs efficiently on large machines.

1.5 Dissertation Outline

Chapter 2 introduces the Charm++ parallel programming language, the features that make it well suited for programming very large parallel machines, and the advantages from automatic load balancing. Chapter 3 presents the design of the BigSim parallel emulator, its performance and its utility. In Chapter 4, we present the PDES techniques we used for extending the emulator for predicting parallel performance of applications on very large parallel machines. We demonstrate the techniques used by CHARM++ to help writing applications with million-way parallelism in Chapter 5. We also present performance study via simulation of a molecular dynamics simulation program and a Finite Element Methods simulation program in the same chapter. Chapter 6 presents an overview of Charm++ load balancing framework and its load balancing methodology for dealing with dynamic load balancing problems, and Chapter 7 presents the performance optimization techniques in centralized load balancing with sophisticated load balancing strategies. Chapter 8 discusses the limitations of current centralized and distributed load balancing strategies on peta-scale parallel machines, and presents a new hybrid load balancing strategy that combines the benefits of the centralized and distributed strategies. Chapter 9 gives the conclusion of the thesis and discusses the future work.

Chapter 2

CHARM++/AMPI and Processor Virtualization

Charm++ [49] is a parallel programming and runtime system that supports processor virtualization [53, 52]. In Charm++, an application divides a problem into a large number of components (N) (implemented as C++ migratable objects or user-level threads) that will execute on P processors and act as the *virtual processors*. N is independent of P though ideally $N >> P$. The basic mechanism in processor virtualization that enables transparent migration of data is to provide applications a *location independent* view of the world. In a Charm++ application, the user's view of the program consists of these components and their interactions; the user need not be concerned with how the components map to processors. The underlying run-time system takes care of this and any subsequent remapping (see Figure 2.1).

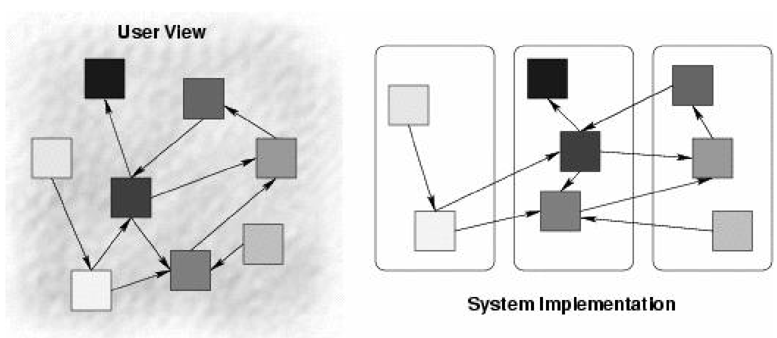


Figure 2.1: Virtualization in CHARM++

In Charm++, these components are known as *chares*. Chares are C++ objects with methods that may be invoked asynchronously from other chares. Since many chares can be mapped to a single processor, Charm++ uses *message-driven execution* to determine which chare executes at a given time.

Objects or chares that carry application code are location independent. Hence chares can migrate from processor to processor freely. One application of migratable objects is load balancing. Objects can migrate from overloaded processors to underloaded processors to achieve better load balance. In order to migrate an object, one needs to pack the data of the object into a message. The message is sent to another processor where the data is unpacked and the object is restored. The Charm++ PUP framework [45] was designed to describe the in-memory layout of an object. The object migration based on the PUP framework can be extended into broader usage, such as migrating an object to disk at runtime for out-of-core execution. In the checkpointing scenario, when an object is checkpointed, it is simply packed and migrated to another location (memory or disk).

Adaptive MPI, or AMPI [42] is an MPI implementation and extension based on Charm++. AMPI implements virtual MPI processes, or VPs, using migratable user-level threads, several of which may be mapped to a single physical processor. AMPI supports adaptive load balancing by migrating MPI threads. Several schemes of fault tolerance support are implemented in Charm++ and AMPI. They range from a synchronous on-disk checkpoint/restart mechanism [41] and a scalable in-memory checkpoint scheme[94], to an automatic fault-tolerant protocol for massively parallel systems [25]. MPI is a special case of AMPI when exactly one VP is mapped to a physical processor.

2.1 Charm++ vs. Conventional Programming Models

Conventional models such as message passing, shared memory and data parallel programming need a lot of programmer effort to efficiently utilize a peta-scale machine 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 peta-scale machines. 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 [65].

For a peta-scale 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 read-only 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.

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 [21]. 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. In Charm++, load balancing can be handled by the run-time system, without changing the programmer's view of his applications. Thus, only the load balancing run-time needs to be changed for a peta-scale machine, which can simultaneously improve performance and reduce programmer effort. Chapter 6 will describe the CHARM++ load balancing framework in more detail.

Load balancing on peta-scale is more difficult than load balancing conventional machines. As an example, when 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 due to 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 order for load balancing run-time to work efficiently, an application needs to be decomposed into more fine-grained tasks than the number of processors. One of the questions that arises while programming peta-scale machines 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. In Chpater 5, we will demonstrate the usefulness of the object model in creating an application with million-way parallelism.

Chapter 3

Dealing with Very Large Parallel Machines

Parallel machines with tremendous computation power are being built with surprising speed. However, it is very difficult to understand how an application might behave on such large machines, and it may be hard to validate the correctness of a parallel algorithm without actually running the application.

In this chapter, we explore the idea of using a parallel emulator to mimic a class of peta-scale machines and support very large scale parallelism. We then use Charm++ as an example of utilizing the emulator for supporting high level parallel programming models for peta-scale machines.

Emulating a peta-scale machine with millions of processors on a much smaller parallel machine with limited resource presents many challenges. We explore the idea of applying processor virtualization with user-level threads in CHARM++ run-time on a scalable efficient parallel emulator. The emulator we present here also represent our first step towards a parallel simulator that is capable of predicting performance of parallel applications. We will present our parallel simulator in Chapter 4.

3.1 Need for Emulator of Very Large Parallel Machines

An emulator, by definition, is a device, computer program, or system that accepts the same inputs and produces the same outputs as a given system. In our case, the purpose of an emulator is to emulate a parallel message passing application as if it executed on a very large parallel machine. It does not have to produce performance information about the run.

An emulator for a peta-scale machine is necessary especially when the target machine is not yet operational. Even when the machine is operational, the access to the machine may be restricted and compute time is difficult to get in order to debug parallel applications. Thus, an emulator provides a nice way to debug a parallel application and validate the algorithm offline.

3.2 Methodology

Emulating the execution of a parallel program on a machine with tens of thousands of processors is not trivial. Directly running an application, for example with:

"**mpirun -np 64000 ./app**"

on a small cluster certainly is infeasible due to the limitations of operating systems, such as, on the number of processes allowed to be created. In order to emulate such a machine, one processor of a host machine needs to emulate multiple target processors.

To develop such a large scale emulator, we extended the idea of *processor virtualization* into a two-level virtualization. At user level, a program computation is divided into a large number of chunks (so called virtual processors), which are mapped to simulated processors by an adaptive, intelligent runtime system. At emulation level, these simulated processors are mapped to actual physcical processors by the same run-time system. The simulated processors are represented by CHARM++ user-level migratable threads, which are treated as another level of virtualization. This approach allows us to exploit the powerful run-time system of CHARM++ to achieve high scalability of the emulation.

We designed and implemented a parallel multi-threaded emulator called **BigSim Emulator** that mimics *low level* hardware and message passing primitives to facilitate the execution of very large scale parallel applications. The low level functional API provided by the emulator is designed to be general enough for a wide variety of parallel machines. With such low level primitives, a user can write parallel applications running on the emulator. However it requires a lot of effort in writing such applications. Thus, the need for high productivity requires high level parallel programming languages. We demonstrate the use of emulator for developing high level languages using Charm++ and AMPI as examples.

3.3 Parallel Emulator

In order to emulate a future massively parallel machine on an existing parallel machine with only hundreds of processors, one physical processor has to emulate hundreds or even thousands of processors of the target machine. One question arises - is it feasible to emulate an application on machines with such scale in terms of memory and CPU cost? We believe the emulation is still possible because:

- Although the overall compute power of peta-scale machines is enormous, they are likely constructed based on compute nodes with modest performance. Thus it is feasible to conduct the simulation for a short run.

- Some planned machines such as BlueGene/C have low memory-to-processor ratio. BlueGene/C was originally designed to have about half a terabyte of total memory. However, each chip with 25 processors shares only 12MB of memory. Thus, to emulate BlueGene/C running an application which uses the full machine will require "just" 500 processors of a traditional parallel machine with 1GB memory per processor.

- For planned machines with high memory-to-processor ratio, no existing parallel machine has enough memory to simulate the full machine. However, many real world applications such as molecular dynamics simulation do not require a large amount of memory.

- For applications that do require a large amount of memory, it is still possible to use automatic out-of-core execution [73] to temporarily move the data in memory to disk when it is not needed immediately. This swapping slows down the simulation. However if the only thing we are interested in is the *predicted* running time which need not be affected by swapping time, for a few timesteps, this slowdown is still acceptable.

In order to support high degree of parallelism, we designed a low level abstraction of peta-scale machines to provide access to a machine's capabilities. In the programmer's view, each node consists of a number of hardware-supported threads with common shared memory. A runtime library call allows a thread to send a short message to a destination node. The header of each message encodes a handler function to be invoked at the destination. A designated number of threads continuously monitor the incoming buffer for arriving messages, extract them and invoke the designated handler function. We believe this low level abstraction of the peta-scale architectures is general enough to encompass a wide variety of parallel machines with different numbers of processors and co-processors on each node.

We have developed a software emulator [76] based on this low level abstraction. The emulator can be easily configured for specific target machines. Figure 3.1 describes the functional view of an emulated

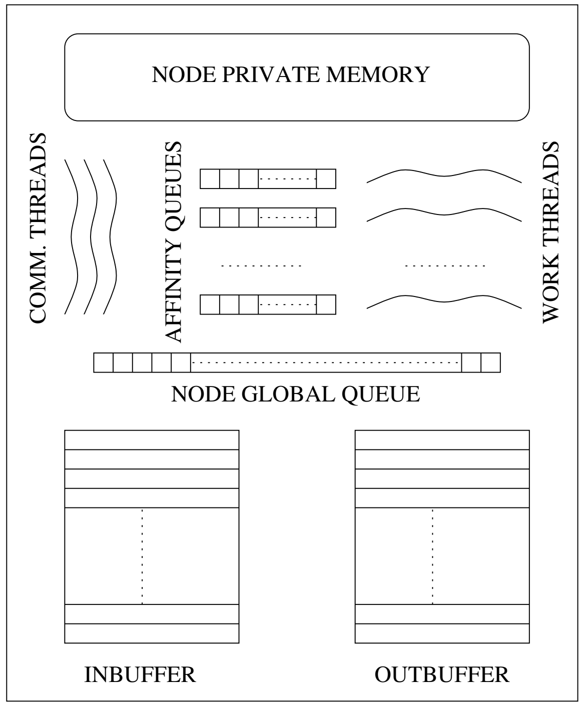


Figure 3.1: Functional view of an emulated node

node. Within a node, we divide threads into worker threads and communication threads. Communication threads check 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.

The emulator provides a function API for setting up the emulated machine configurations and providing functions for message passing. The details of the API definitions are described in next section.

3.3.1 Emulator Function API

Emulator API includes the following function definitions:

- *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 returns a globally unique identifier associated with it.

- *void BgAddMessage(int threadID, int handlerID, int nBytes, char *msg)* — is called to schedule a message for execution on the local target node specified by the threadID.

- *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.

3.3.2 Emulating Hardware Processors

The number of processors of a target parallel machine typically is far larger than the size of the simulating parallel machine, thus multiple processors of target parallel machines are emulated using multiple flows of control running on a single simulating processor. A flow of control is a single sequential stream of executed instructions, including subroutine calls and returns. The performance and limitations of such flow of control mechanisms determine if the emulation of a petascale machine is feasible or not.

Many mechanisms are available for supporting multiple flows of control. We evaluated three specific mechanisms: processes, kernel-space threads and user-space threads.

- OS-level Processes

  Process is the simplest and oldest flow of control. It wraps a complete address space around a flow of control. The drawback of using processes for emulation is that processes are considered "heavy-weight": it is expected that the substantial amount of process kernel state increases the amount of memory used by each process, increases the overhead of process creation and switching, and leads to hard limits on the number of processes that can be created in a user program.

- Kernel-space Threads

  Kernel threads consist of a set of registers, a stack, and a few corresponding kernel data structures.

Unlike processes, threads within a process all share the same address space. All modern machines support kernel-space threads, most often via the POSIX threads interface (known as *pthreads*).

Kernel-space threads often are implemented in the kernel using several tables (each task gets a table of threads). In this case, the kernel schedules each thread within the timeslice of each process. There is a little more overhead with mode switching from user → kernel → user and loading of larger contexts. The advantages of kernel-space threads over processes is faster creation and context switching.

- User-space Threads

  User-space threads avoid the kernel and manage the tables themselves. Often this is called "cooperative multitasking" where the task defines a set of routines that get "switched to" by manipulating the stack pointer. Typically each thread "gives-up" the CPU by calling an explicit *yield* call, sending a signal or doing an operation that involves the switcher. User threads typically can switch faster than kernel threads. [1] Many implements of user-level threads exist, including Gnu Portable Threads (GNU Pth [4]), and Converse threads (*CthThreads*).

  Converse threads, or CthThreads, are based on various thread packages and implementations that depend on what is available on the platform. CthThreads include QuickThreads [62] [2], context threads based on *getcontext/setcontext* system calls [3], Windows NT Fiber, etc.

User-space threads in general have several advantages over the other two flow of controls in the implementation the emulator for very large number of processors. First, the usage of processes and kernel-space threads is limited by various system constraints. As illustrated in Table 3.1 with some of the experiments conducted by Orion Lawlor in our group, both processes and pthreads creation are often limited by system configuration and kernel, for which a normal user often does not have control. Our experiments have shown that there is wide variation in the limitations and the performance of these methods on different machines. For example, an unmodified Linux Red Hat 9 machine can spawn no more than 256 pthreads; meanwhile the per-user process limit on an IBM SP was only 100 processes. By contrast, we could create tens of thousands of user-level threads easily on all platforms.

[1] however, Linux kernel threads' switching is actually quite close in performance.

[2] QuickThread was first implemented by David Keppel, it was further extended by various members of our group including myself.

[3] Context thread based CthThreads were implemented by myself.

Flow of control	Limiting Factor	Linux	Sun	IBM SP	Alpha	Mac OS	Itanium
Process	ulimit/kernel	4000	5000	100	1000	500	50000+
Kernel Threads	kernel	250	3000	2000	50000+	7000	30000+
User-level Threads	memory	50000+	20000+	10000+	20000+	15000+	50000+

Table 3.1: Approximate practical limitations (on stock systems) for various methods to implement flow of control.

Second, stack size usage of the threads or processes has a great impact on the emulation if the emulated system is very large. As the simple calculation in Table 3.2 reveals, an unnecessarily large stack size can increase the total memory footprint dramatically, especially for a small simulating cluster with only limited number of nodes available. In general, it is infeasible or non-portable to set stack size for a process or pthreads [4]. However, user level threads allow a user to easily control the stack size of each thread. In practice, we found that a 10KB stack size is sufficient for most emulations that do not allocate big data structure on stack.

	Number of Emulated Processors				
Stack Size per Thread	16K	32K	64K	128K	256K
64KB	1GB	2GB	4GB	8GB	16GB
128KB	2GB	4GB	8GB	16GB	32GB
1024KB	16GB	32GB	64GB	128GB	256GB

Table 3.2: Total memory needed for a number of emulated processors with various stack sizes of a control flow

Third, the context switch overhead is an important factor in the emulator efficiency. We examined the context switch performance of three different implementations of flow of control:

1. Processes, created using fork() and yielding using sched_yield().

2. Pthreads, created using pthread_create() and yielding uysing sched_yield().

3. Converse threads, user level threads created using CthCreate() and scheduled using CthYield().

We run tests on a variety of platforms including Linux (Figure 3.2), Sun Solaris (Figure 3.3) and Mac OS (Figure 3.4). Our experiments show that our Converse user level thread implementation generally is much more efficient in context switching performance.

[4] pthread_attr_setstacksize() can set pthread stack size, however it is not supported on some platforms such as the IA64

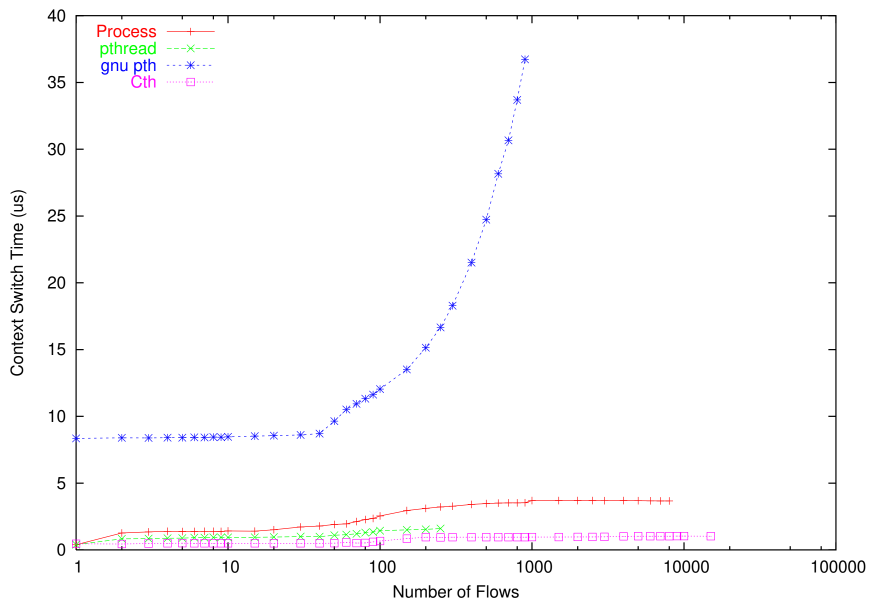


Figure 3.2: Context switching time vs. number of flows on a x86 Linux machine.

In summary, using user level threads (CthThreads) to emulate hardware processors has the following advantages over processes and kernel-space threads:

1. CthThreads are highly portable. Currently, they run on all popular parallel systems including Blue Gene/L and even on Windows PC.

2. Using CthThreads avoids potential system limitations that processes and kernel threads may have on the number of flows of control allowed to create.

3. Thread stack is explicitly managed in the CthThreads implementation; reducing an unnecessarily big stack size helps reducing memory usage in the emulation.

4. CthThreads generally have faster context switching time. This improves the emulation performance significantly because an emulation normally involves a very large number of target processors and threads emulating them.

Ideally, a standard API and a skeleton for user-level threads would be supported in the future by OS and compilers.

In [76], we demonstrated that using CthThreads our emulator enables us to model Blue Gene/C nodes organized in a 34 x 34 x 36 grid with 200 threads per node (i.e. 8M threads) on 96 physical processors of

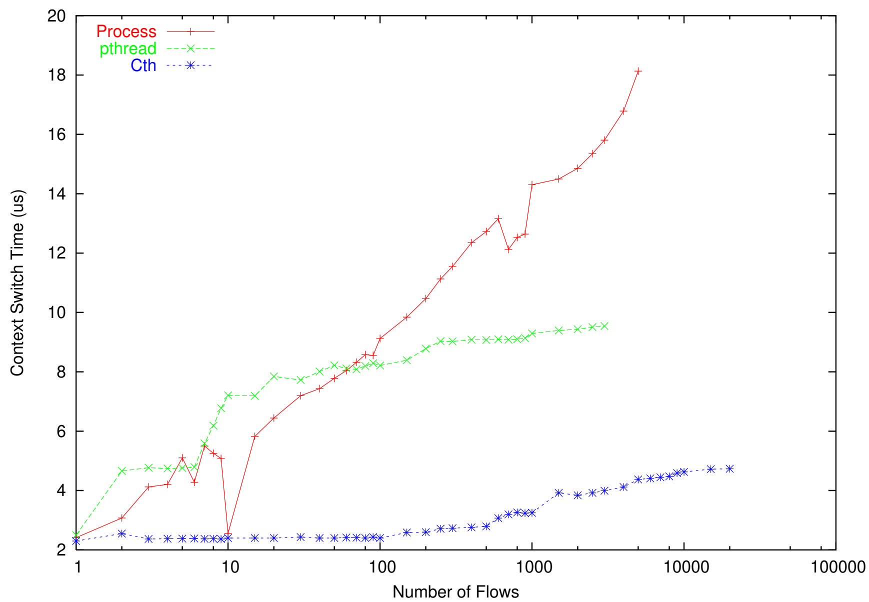


Figure 3.3: Context switching time vs. number of flows on Sun Solaris machine.

ASCI-Red. This involves creating approximately 80,000 user-level threads per ASCI-red processor.

3.3.3 Emulator Performance

In this section, we will present two case studies. The first case study is on the emulation performance using user level threads (CthThreads) compared with the kernel threads (pthreads) in emulator implementation to illustrate the advantage of using CthThreads. The second performance study demonstrates the scalability of our emulator.

Performance Comparison between CthThreads and Pthreads

To demonstrate the performance benefit of using CthThreads, we compared the performance of emulation using Pthreads and CthThreads to emulate target processors.

We ran an emulation of a parallel Jacobi program on one physical processor of a SUN machine at CSAR [2] using CthThreads and Pthreads and compared the total time taken for the emulation. Since the number of pthreads one can create on a SUN machine is only about 3000 (given stack size of 128KB), in the experiment we only emulated target machines of up to 2000 processors.

The result is shown in Table 3.3. The emulation time shown in the table is the total time spent on the whole emulation, including the context switching time. As can be found in Figure 3.3, when there are 2000

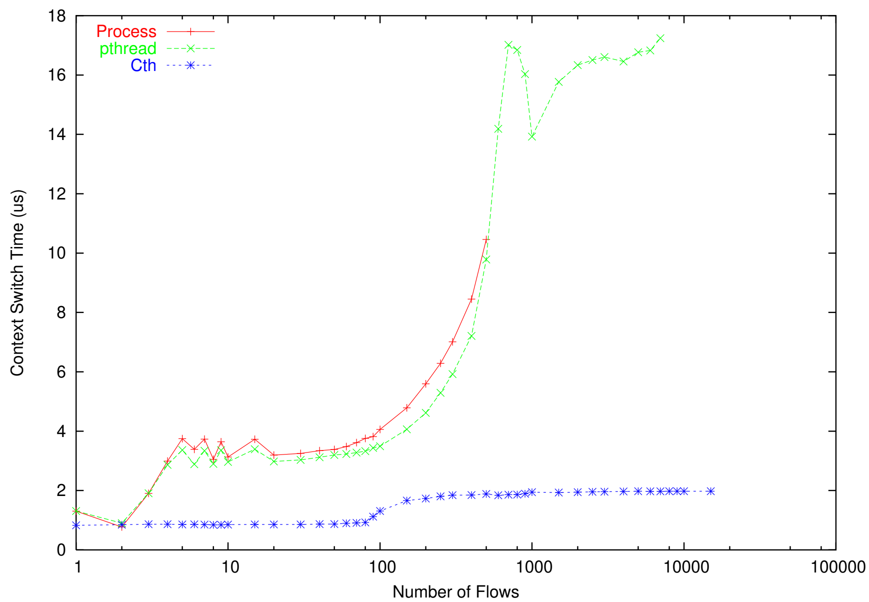


Figure 3.4: Context switching time vs. number of flows on Mac machine.

Number of PEs emulated	250	500	1000	2000
Pthreads	7.73	9.59	13.95	30.42
CthThreads	3.33	4.49	7.4	15.67

Table 3.3: Emulation time (in seconds) using CthThreads and Pthreads with BigSim emulator

flows of control, the CthThreads context switching time is about 3.8us, and the Pthreads context switching time is about 9.5us. The difference in context switching cost leads to significant performance improvement in emulation achieved by using CthThreads.

Parallel Emulator Scalability

Our BigSim emulator demonstrates good scalability. We ran a molecular dynamics (MD) simulation code — LeanMD — to emulate a Blue Gene like machine with 200,000 processor on LeMieux at PSC [5]. We measured the time taken to emulate one timestep of the MD simulation using 4 to 64 LeMieux processors. The result is shown in Figure 3.5, which demonstrates excellent scalability of the emulator in this test. In fact, there is a superlinear speedup of 67 over 64 processors (normalized by the 4 processor time). This is due to a smaller memory footprint per physical processor as the number of processors is increased, with corresponding caching effects.

[5] LeMieux is 750 Quad Alphaserver ES45 node machine at Pittsburgh Supercomputing Center (PSC)

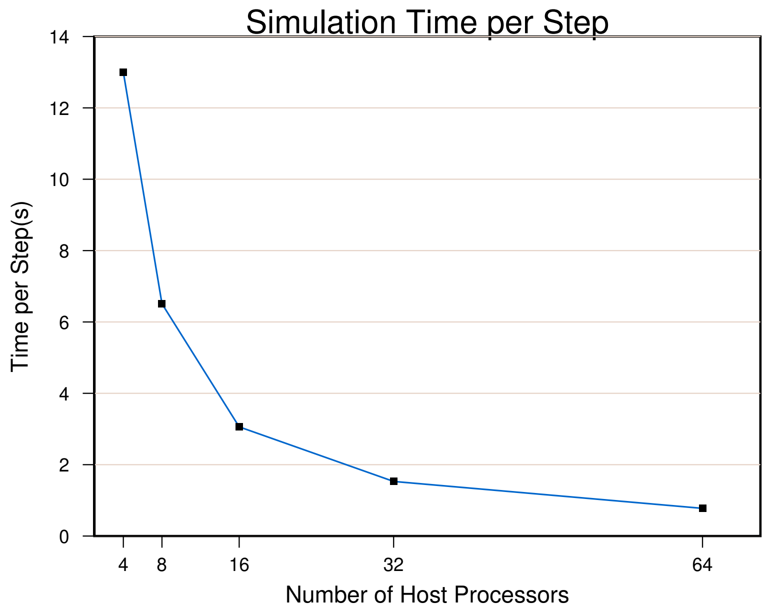


Figure 3.5: Simulation Time per Step

3.4 Porting High Level Parallel Languages to the Emulator

We implemented the CHARM++ programming model on the BigSim emulator to enable the study of performance and scalability of a peta-scale machine [95]. In short, we call this implemenation of CHARM++ as **BigSim** CHARM++. In this section, we describe some design issues involved in the implementation of CHARM++ on top of the emulator, along with some optimizations to improve the efficiency of CHARM++ and the emulator.

3.4.1 Design Issues

In CHARM++, the programmers do not 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 normal 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. For CHARM++ on BigSim emulator, we considered two alternative strategies to guarantee atomicity.

1. Treating an entire emulated node (with many processors and possibly hundreds of threads) as a single CHARM++ processor. When a message is sent to a CHARM++ processor, the emulator's 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 peta-scale machine node are mapped to a single emulator processor, where there is no preemption.

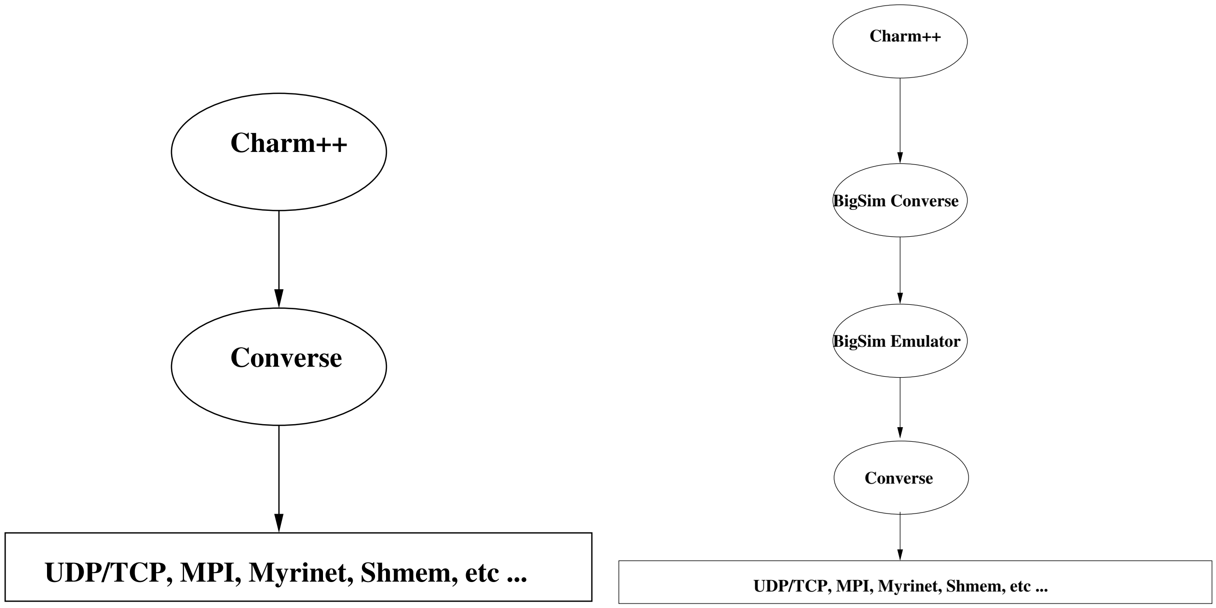
2. Treating each worker thread in an emulator as a CHARM++ processor. Chare objects are anchored to individual threads. When a message arrives, it will be scheduled into the affinity queue of the worker thread specified by the message. This approach may require explicit load balancing of work among threads of a node.

Both schemes are implemented in CHARM++ on the emulator. However, the first scheme has an advantage that it needs a smaller amount of memory for per-processor data structures maintained by the CHARM++ run-time system.

CHARM++ is implemented using the Converse layer as shown in Figure 3.6(a). 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.

For BigSim CHARM++, we need a Converse layer (BigSim Converse) that is implemented on the BigSim emulator API, to provide the same functionality of the original Converse. However, the emulator that this BigSim Converse is built on is implemented itself upon the original Converse. Figure 3.6(b) illustrates the new BigSim CHARM++ hierarchy. It shows that the Converse layer is used in two contexts: in one place to represent the real machine — Converse; and in the other place to represent the virtual emulated machine — BigSim Converse.

While the two uses of 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 BigSim Converse, Processor Private Variables are private to each BigSim emulated node. This



(a) Original CHARM++ hierarchy

(b) BigSim CHARM++ hierarchy

Figure 3.6: CHARM++ hierarchy

causes significant name conflict problems.

To solve these problems, 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 BigSim emulator. A similar implementation, consisting of simple wrappers, was created for normal Converse as well. Both of these were encapsulated in different *name spaces*. This layered implementation is shown in Figure 3.7. Using C++ namespaces, the CHARM++ runtime can now easily switch between the BigSim Converse and the normal Converse. This greatly simplifies the implementation while allowing two versions of CHARM++ to coexist in one system. This is an example of what a routine in the BigSim Converse version looks like:

```
namespace BGConverse {
 ...
 void
 CmiSyncSendAndFree(int pe, int numBytes,
                              char *msg)
 {
  int x,y,z;
```

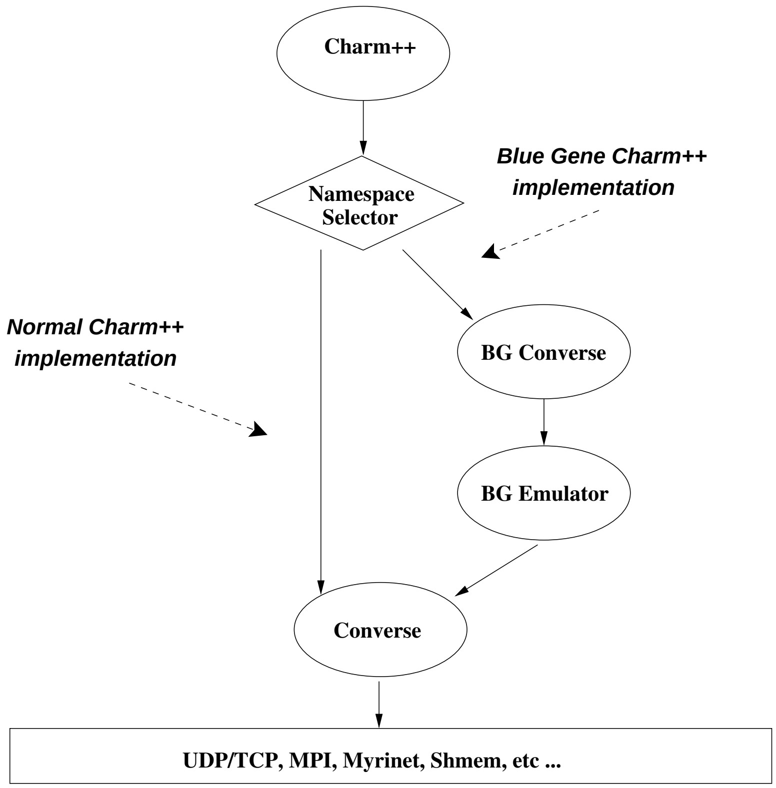


Figure 3.7: Layered Implementation of Blue Gene CHARM++

```
    // 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, we created an implementation of BigSim CHARM++ that shares the code base with the rest of the CHARM++ RTS. With BigSim CHARM++, applications originally written for CHARM++ can be easily ported to the BigSim CHARM++ running on the emulator.

3.4.2 Experience with the Emulator

The emulator has shown to be very useful for discovering practical constraints such as memory or communication bottlenecks by emulating an application on a very large scale target machine using a much smaller parallel machine. It is also helpful for identifying performance bottleneck in run-times and applications during optimization.

As an example, when we simulated an FEM application on a Blue Gene like machine, we exposed a number of practical bottlenecks to execution on very large machines. These include a limitation of METIS mesh partition tool that failed to partition even a 4K element mesh, and a memory usage bottleneck in AMPI implementation. The detail of these discoveries is presented in Section 5.2.

Simulating load balancing on peta-scale machines also helps us discover a number of memory bottlenecks. For example, we found GreedyCommLB, a greedy centralized load balancer that takes communication into account, uses memory of $numPes * numObjs$ doubles, where $numPes$ is the number of processors and $numObjs$ is the number of objects in the application. For example, load balancing an application that has 128K objects running on 32K processors requires 32GB of memory on the central node! The solution was to avoid the matrix and use a minHeap whose size is only proportional to the $numObjs$.

Our experience shows that the emulator is very useful in helping developers validate the parallel algorithms in both the run-time system and applications so that they can be debugged and ready to run on a very large parallel machine even before the machine is available. As an example, with off-line debugging and tuning, CHARM++/AMPI porting to IBM Blue Gene/L was done in a day with access to the real machine.

Chapter 4

Performance Prediction for CHARM++/AMPI Applications

The emulator discussed in the previous chapter is useful only for studying programming models and application development issues that arise in the context of peta-scale machines. Specifically, the emulator does not provide detailed performance information.

Developing a simulator for running a large scale application on peta-scale machines can be useful in many ways. From an architecture designer's point of view, it would help to reveal how target applications might perform on large machines for certain machine parameters. If we could simulate how the application would behave on the large machine, we might be able to improve the design of the machine before it is built.

For an application programmer, it is often difficult to predict what effect modifications to the algorithms will have on performance. Having a performance modeling tool, one could predict, identify performance bottleneck and optimize the performance of the application so that it is ready as soon as the machine is available. Even for existing large machines, our performance prediction approach is useful. For example, consider the process of performance optimization of a complex application. Time on large machines is hard to get and must be reserved well ahead of time. A performance measurement run typically takes only a few minutes. However, every time such a run is carried out, application developers must spend some time visualizing and analyzing the performance data before the next set of optimizations are decided upon, at which point one must wait in queue for the next running slot (typically at least for hours or a day). With a simulator, this performance debugging cycle can be shortened considerably. Even if predictions are approximate, they are still useful for evaluating the algorithms to identify major performance issues such as load imbalance, serialized bottlenecks etc.

4.1 Methodology

To evaluate planned peta-scale machines, it is essential to be able to evaluate the performance of applications on the machines before they are built. However, parallel performance is notoriously difficult to model. It is clearly not adequate to multiply the peak floating point performance of individual processors by the number of processors. Performance of communication subsystems, complex characteristics of target applications during their multiple phases, and the behavior of run-time support systems interact in a nonlinear fashion to determine overall performance. For example, for a particular problem size, communication latencies may be completely masked by effective overlap of communication and computation by the application. However, for the same application, if the ratio of computation to communication decreases below a threshold, the performance may be totally dominated by the communication performance.

Essentially, there are two methods of traditional performance modeling. One method is analytical performance modeling [55, 27], and the other is simulation-based performance modeling [6].

Most analytical performance modeling schemes are based on certain types of analytical model based on certain performance metrics, and often require training data from runs on the target machines. Analytical performance modeling is not the interest of this thesis. Instead, we adopted the simulation-based performance modeling which is commonly used for performance studying of complex systems. In this method, to make performance predictions, one can either (a) record traces during emulation, and then run a sequential trace-driven simulation or (b) modify the emulator to carry out a *Parallel Discrete Event Simulation* (PDES).

The sequential trace-based approach is infeasible if the trace data will not fit in the memory of a single processor, or will take too long to run. To improve the simulation efficiency, parallel PDES execution [33, 75] is explored to reduce the execution time of simulations as an alternative to the sequential execution. However, parallel execution further complicates the simulation because a parallel simulator needs to use certain protocols to synchronize the simulation processes in order to maintain simulation accuracy. This approach appears to be quite challenging, in view of the failure of typical PDES systems to provide effective parallel performance [37]. However, we have developed a scheme in **BigSim Simulator** [93, 96, 97], an extension of BigSim Emulator with performance prediction capability, that makes this approach feasible and efficient by exploiting the *inherent determinacy* of parallel programs.

4.2 Related Work in PDES

Complex models may often involve components that are distributed over a network and generally require significant execution times. Simulation of such systems may be beyond the capability of a sequential computer, and require a parallel computers with significant compute power and memory capacity.

The field of parallel and distributed simulation has grown over the past fifteen years to accommodate the need of simulating complex models using a distributed rather than sequential method. *Parallel discrete event simulation* (PDES) is a popular technique that assumes the system being simulated only changes state at a discrete points in simulated time. It has been extensively studied in literature [44, 30, 84].

As an example, in a simulation of an application running on a parallel machine, the simulation entities can include simulated processors, network components and all software components in the user application such as processes in MPI or parallel objects in CHARM++. The target processors are then mapped to logical processors (LPs), each of which has a local virtual clock that keeps track of its progress. In the simulation, user messages together with their subsequent computations play the role of events.

4.2.1 Synchronization Protocols

In the parallel simulation, each LP works on its own by selecting the earliest event available to it and processing it without knowing what happens on other LPs. Thus, methods for synchronizing the execution of events across LPs are necessary for assuring the correctness of the simulation.

Two broad categories of synchronization strategies are *conservative* approach and *optimistic* approach.

In conservative synchronization approach, one has to ensure the safety of processing the earliest event on an LP in a global fashion. The drawback of this method is that the process of determining safety is complex and expensive. It also reduces the potential parallelism in the simulation. The conservative synchronization approach is used in many simulators including Wisconsin Wind Tunnel (WWT) [75], LAPSE [33] and MPI-SIM [74].

Optimistic synchronization protocols allow LPs to process the earliest available event with no regard to safety. When causality errors occur, a rollback mechanism is needed to undo earlier out-of-order execution and recreate the history of events executed on the LP as if all events were processed in the correct order of occurrence. Despite the cost of synchronization, the optimistic approach exploits the parallelism of simulation better by allowing more concurrent executions of events. In BigSim, we adopt an extension of the optimistic synchronization approach.

The most extensively studied optimistic mechanism is Time Warp, as used in the Time Warp Operating System [44]. Time Warp was notable because it was designed to use process rollback as the primary means of synchronization.

The Georgia Tech Time Warp (GTW) [30] system was developed for small granularity simulations such as wireless networks and ATM networks. One of the GTW features that carries over into BigSim is the *simulated time barrier*, a limit on the time into the future that LPs are allowed to executed.

The Synchronous Parallel Environment for Emulation and Discrete Event Simulation (SPEEDES) [84, 85] was developed with a different optimistic approach to synchronization called *breathing time buckets*.

Á la carte [1] is a Los Alamos computer architecture toolkit for extreme-scale architecture simulation. It uses a conservative synchronization engine, the Dartmouth Scalable Simulation Framework (DaSSF) [69], for the handling of discrete events. They have targeted on simulating thousands of processors. In the Quadrics network simulation of SWEEP3D [17], they reported no speedup when adding more computational nodes in the simulation. The only performance data they reported in the paper are only for simulating a 36-process run of SWEEP3D using 2 to 36 real processors.

Conservative simulators require a lookahead, which imposes a high global synchronization overhead, and typically limits the amount of parallelism one can exploit in simulation. On the other hand, the optimistic general purpose simulators, when directly applied to performance prediction of parallel applications on extreme scale of processors (tens of thousands or even millions of processors), may lead to very high synchronization overhead, caused by the need to rollback a simulation when a causality violation is detected.

The synchronization overhead of optimistic concurrency control consists of:

- **Checkpointing overhead** - time spent in storing program state before an event is executed which might change that state.
- **Rollback overhead** - time spent in undoing events and sending cancellation messages.
- **Forward execution overhead** - time spent in re-executing events that were previously rolled back.

4.2.2 Performance Prediction for Parallel Applications

All of the important behaviors that affect a parallel application on a very large parallel machine can be efficiently described as actions occurring at a particular time and lasting for a certain duration. These behaviors are thus best simulated with a parallel discrete event model.

One approach of such simulation is *direct-execution* simulation [75, 6, 74]. Direct execution simulators use available hardware resources to directly execute the application code to simulate architectural features of interest. Á la carte [1], a Los Alamos computer architecture toolkit for extreme-scale architecture simulation, uses a conservative synchronization engine for simulating thousands of processors.

MPI-SIM [74] is a library for execution-driven parallel simulation of task and data parallel programs. It simulates user level code using a direct execution method and its main focus is evaluating the performance of parallel programs. Adve et al. [6] optimized the simulation by compiler techniques that help to avoid executing large portion of computational code by identifying the subset of the computations whose values have no significant effect on the performance. MPI-SIM is one of the components of a bigger project — COMPASS [10] which is a portable, execution driven, asynchronous simulator that is capable of predicting the performance of larg-scale parallel programs, including computation and I/O intensive applications. The simulation kernel of COMPASS provides support for both sequential and parallel execution of the simulation. Parallel execution is supported via a set of conservative synchronization protocols. The interconnection network model used in the simulation kernel ignored contention in the network.

To our best knowledge, there is no parallel performance simulator that adopts the optimistic synchronization protocol due to its complexities and possible overheads incurred. While the conservative synchronization protocol is relatively simple in avoiding casuality errors, it inherently has limited ability to agressively exploit large scale parallelism involved in simulating extremely large applications. In this chapter, we present our research results in adopting an optimistic synchronization protocol used in our parallel performance simulator to reduce synchronization overhead.

4.3 Component Performance Models

Converting our emulator to a simulator requires correct estimation of the time taken by sequential code blocks and messaging. There are a range of possible methods for prediction with different degrees of accuracy.

Predicting the Time of Sequential Code

The wallclock time taken to run a section of code on a traditional machine cannot be directly used to estimate the compute time on the target machine. Because we do not know the time taken for a given sequence of instructions on the target machine, we can use a heuristic approach to estimate the predicted computation time on the simulator. Many possible methods are described below. They are listed in increasing order of accuracy (and of complexity involved).

1. User-supplied expression for every block of code, estimating the time that it takes to run on the target machine. This is a simple but highly flexible approach, since the callback API for this purpose can be used to implement more sophisticated methods.

2. Wallclock measurements of the time taken on the simulating machine can be used via a suitable multiplier (scale factor), to obtain the predicted running time on the target machine.

3. A better approximation is to use hardware performance counters on the simulating machine to count floating-point, integer and branch instructions (for example), and then to use a simple heuristic considering the time for each of these operations on the target machine to produce the predicted total computation time. Cache performance and memory footprint effects can be approximated by percentages of memory accesses and cache hit/miss ratios.

4. A much more accurate way to estimate the time for every instruction is to use a hardware simulator that is cycle accurate for the target machine.

The first three of the methods are currently supported in our simulator.

Predicting Network Performance

It is also necessary to simulate the network environment of the target machine to get the accurate performance prediction. The possible approaches are described below in increasing order of accuracy (and of complexity).

1. No contention modeling: The simplest approach ignores the network contention. The predicted receive time of any message will be just based on topology, designed network parameters and a per-message overhead.

2. Back-patching: this approach stretches communication times based on the communication activity during each time period, using a network contention model.

3. Network simulation: This approach uses detailed modeling of the network, implemented as a parallel (or sequential) simulator.

The first and third methods are supported in our simulator. Although the no contention modeling is not as accurate as the detailed network simulation, it is still useful for simulating applications that network contention is not a significant concern. The detailed network simulation is far more expensive than the no contention modeling as well.

4.4 Parallel Simulator

BigSim simulator adopts a parallel discrete event simulation (PDES) methodology with an optimistic approach. The simulation involves letting the emulated execution of a proram proceed as usual, while concurrently running a parallel PDES engine which corrects time-stamps of individual messages.

As discussed in Section 4.2, an optimistic approach of simulation incurs considerable synchronization overhead, which may be prohibitive given the size of the simulated peta-scale machines. One simple observation of parallel applications, however, leads to optimizations in the BigSim simulator to make such simulation feasible. That is the *inherent determinacy* often found in parallel applications. Those applications tend to be deterministic, with a few exceptions (such as branch-and-bound and certain classes of truly asynchronous algorithms). Parallel programs are written to be deterministic. They produce the same results on repeated execution, and even though the execution orders of some components may differ slightly, they carry out the same computations.

In order to detect and exploit this inherent determinacy, we designed the BigSim simulator to be integrated with the language and run-time to reduce the overhead of synchronization. We have designed and implemented this method for a broad class of parallel applications.

4.5 Simulating Parallel Applications

We classify parallel applications into two categories depending on the degree of their determinancy. They are linear order applications with very strong inherent determinancy, and non-linear order applications with weak determinancy. In the following sections, we discuss strategies for simulating both classes of applications.

4.5.1 Simulating Linear Order Applications

A simple class of deterministic programs are those which permit messages to be processed in exactly one order. We call them *linear order* parallel programs. As an example, consider an MPI program that uses no wildcard receive and ensures only one message with a given tag and sender-processor is available at a time.

In linear order parallel applications, application messages are guaranteed to be delivered to the application in the expected order. The communication runtime handles any out-of-order message by buffering it until the application asks for it. The simulation is trivial in this case since no simulation event needs to be rolled back. Therefore, all overhead of checkpointing, rollback and re-execution can be avoided.

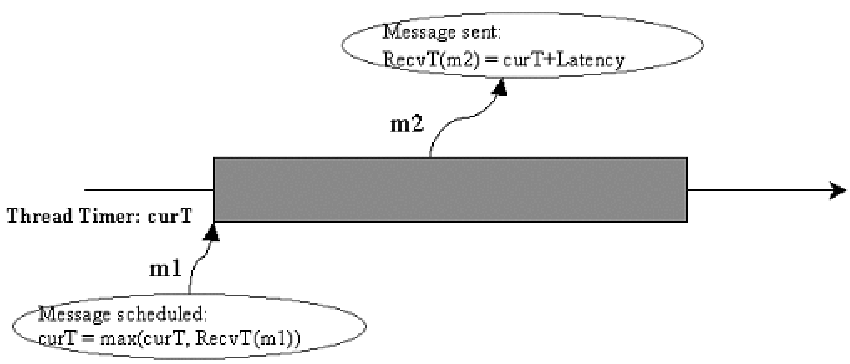


Figure 4.1: Timestamping events

The synchronization algorithm for the simulation of such parallel applications is simple. As illustrated in Figure 4.1, a virtual processor timer ($curT$) is maintained for each logical processor (LP) (implemented as a user-level thread in BigSim). When a message ($m2$) of the program is sent out, we calculate its predicted receive time on the destination LP using a network model (Section 4.3). For example, in the simple model it is just the sum of the current thread time and the expected communication latency to the destination. The predicted receive time is then stored along with the message. When the *receive* statement for this message is executed ($m1$) on the destination LP, its virtual processor timer is updated to the maximum of the current thread time and the predicted receive time of the message. Since there is only one order of execution possible, no rollback is necessary, and thus no checkpoint is needed either.

4.5.2 Simulating a Broader Class of Applications

Linear order applications are limited in their expressiveness. For example, many parallel programs written in message-driven languages such as CHARM++ do not belong to this category. In CHARM++, messages directed to an object can arrive in any order.

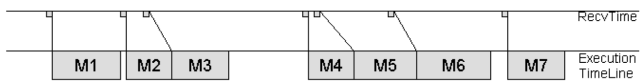
In the next subsection, we first describe a simple class of message-driven applications and their simulation, followed by a more complex and broader class of applications.

Simulation of Message-Driven Programs

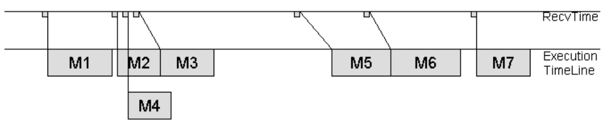
In message-driven programs, the execution of the function associated with a message is ready to be scheduled when the corresponding message invoking it arrives. In *atomic message-driven* programs, the execution is deterministic even when messages (method invocations) execute in different sequences on an object: the object is either providing information to other objects in request-response mode, or processing multiple method invocations that complete as a set before the object (and the application) continues on to the next phase, possibly via a reduction.

Due to the deterministic property in method invocation, the simulation for such class of applications does not require checkpointing and re-executing an event, since re-execution of an event in the time of rollback will only produce the same states. In this case, the rollback process is then simplified as a *timestamp correction* of events in the simulation.

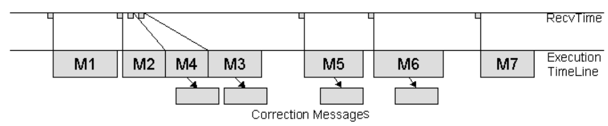
Figure 4.2(a) shows an initial timeline as an example. Each block represents an event in the simulation, recording an execution of an application function associated with the message. The header of each message stores its predicted arrival time (shown as *RecvTime* in Figure 4.2(a)). A message can be executed on an LP if it is idle at the receive time. If the message arrives in the middle of an execution, it has to wait until the current execution finishes (see M5 in Figure 4.2(a)). Assume that M4 has its predicted receive time now updated to an earlier time as shown in Figure 4.2(b). After timestamp correction, the modified timeline is shown in Figure 4.2(c). M4 is inserted back into the execution timeline with updated *RecvTime* and M5 is now executed right at its *RecvTime*. Note that all the events affected in the execution timeline (M4, M3, M5 and M6) also send out timestamp correction messages to inform all the spawned events about the changes in the timestamps.



(a) Initial timeline



(b) Receive time of M4 is updated to some earlier time



(c) Updated timeline

Figure 4.2: Timelines after updating event receive time and after complete correction

Atomic message-driven programs are relatively rare, but they provide an opportunity to present the above timestamp correction approach in a simpler setting. A more complex class of applications, which we call *non-linear* order programs, is a generalization of both atomic message-driven programs and linear order programs. Non-linear order programs have even more complex dependences, yet are essentially deterministic, as illustrated in the next subsection.

Non-linear Order Parallel Applications

Message-driven systems such as CHARM++ allow a process (or object, in the case of CHARM++) to handle messages in multiple possible orders. This enables a better overlap of communication and computation, since the system can handle the messages in the order they are received. However, from the point of view of simulation, this creates a complex causal dependence.

As an example, consider a 5-point stencil (Jacobi-like) program with 1-D decomposition, written in a message-driven language: every chunk of data (implemented as parallel migratable object) waits for a message from its left neighbor and another from its right neighbor. As illustrated in the first timeline of Figure 4.3, the message from the right object invokes the function *getStripFromRight* on this object and the message from the left object invokes *getStripFromLeft*. These two messages generate events $e1$ and $e2$ in the simulator. After both messages arrive, the program calls the function *doWork* to do the calculation. Since the messages from left and right may arrive out of order, both functions need to be written in such a way that the last-invoked function should call *doWork*.

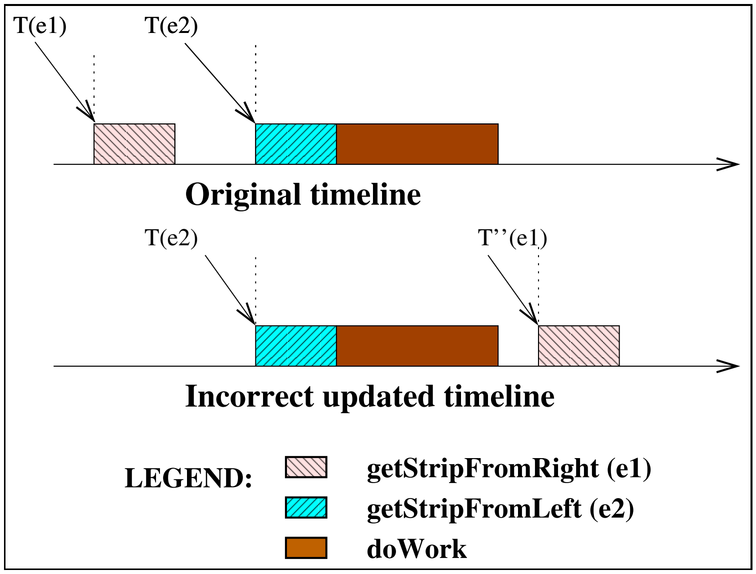


Figure 4.3: Incorrect timestamp correction scheme

When timestamp corrections are performed, the updated receive time of $e1$ may become $T'(e1) > T(e1)$. A naive application of the timestamp correction scheme described in the last section will move event $e1$ to a later point in time, as shown in the second timeline in Figure 4.3. Now the function *doWork* is called by the first arrived message, which is incorrect. This simple example shows that a naive timestamp scheme can easily violate the dependencies among events.

Full checkpointing, rollback and re-execution of events is one (very expensive) solution to ensure the correctness of the simulation. However, at a higher level, the program *is* deterministic — an event corresponding to *doWork* depends on the completion of both *getStripFromLeft* and *getStripFromRight* events, regardless of the order of their arrival. This determinacy of a program can be exploited to carry out the timestamp correction without re-executing the events (i.e. without having to re-execute the application code).

In fact, even from a programming perspective, such dependencies are problematic. Programmers have to write explicit code for maintaining counters and flags (to see if both messages have arrived), and buffering messages. Also, the flow of control is obscured by the split-phase specification style. For these reasons, CHARM++ provides a notation called "Structured Dagger" that allows explicit representation of event dependencies. We propose to extract the dependency information from programs in this language, along with runtime tracking of dependencies, to carry out deterministic simulations without re-execution.

Structured Dagger - a Language for Expressing Event Dependencies: Structured Dagger [47] was developed as a coordination language built on top of CHARM++. It allows a programmer to express the control flow within an object naturally using certain C language-like constructs.

In Structured Dagger, four categories of control structures are provided for expressing dependencies. They are *When-Block*, *Ordering Construct*, *Conditional and Looping Constructs* and *Atomic Construct*.

```
entry void jacobiLifeCycle()
{
  for (i=0; i<MAX_ITER; i++)
  {
    atomic {sendStripToLeftAndRight();}
    overlap
    {
      when getStripFromLeft(Msg *leftMsg)
         { atomic { copyStripFromLeft(leftMsg); } }
      when getStripFromRight(Msg *rightMsg)
         { atomic { copyStripFromRight(rightMsg); } }
    }
    atomic{ doWork(); /* Jacobi Relaxation */ }
  }
}
```

Figure 4.4: Sample code in Structured Dagger

Figure 4.4 shows an example of the parallel 5-point stencil program with 1-D decomposition written in Structured Dagger. In the program, the **for** loop implements the iterations, which begin with calling *sendStripToLeftAndRight* in an **atomic** construct to send out messages to the neighbors [1]. The **overlap** immediately following asserts that the two events corresponding to *getStripFromLeft* and *getStripFromRight* can occur and be processed in any order. The **when** construct simply says that when, for example, *getStripFromLeft* happens, it invokes the action in the **atomic** construct which calls a plain C++ function *copyStripFromLeft* to process the application message. After both events happen, function *doWork* in the last **atomic** construct will be invoked and the program enters the next iteration. Specifically, this sample code describes the event dependencies among events of application message *getStripFromLeft*, *getStripFromRight* and *doWork*.

The Structured Dagger program is compiled and translated into a normal C++ program with parallel runtime function calls. Interacting with the BigSim simulator using function API, the simulator gathers

[1] The atomic construct encapsulates any C language code

dependencies among events to ensure the correctness of the simulation. The new simulation scheme with Structured Dagger for non-linear order parallel applications is described next.

Simulating Non-linear Order Applications: With the help of language and runtime support, our simulator is able to capture the event dependencies which otherwise would be hidden in user programs. During the simulation, the simulator and the CHARM++ runtime system work together to maintain the order of the executions according to the dependencies among events.

This approach also applies to a large class of MPI programs that use MPI_Irecv and MPI_Waitall as well: the *waitall* operation is simply recorded as having backward dependencies on all the pending *irecvs*. To make this happen, the runtime implementation of the MPI calls needs to interface to the simulator with the event dependency information.

In the simulation of both MPI and CHARM++ programs, with the new scheme, the rollback process becomes greatly simplified. No event needs to be re-executed because re-execution of an event would produce the same states. Thus, checkpointing of the states is also avoided since they are not needed any more. Rollback in the new scheme is now simply an extension of the timestamp correction described in Section 4.5.1 under the constraints of the event dependencies. To illustrate the new simulation scheme better, we use the same Jacobi example in Figure 4.4 in the next section.

4.5.3 Implementation Details and Optimizations

We modified the structured dagger coordination language to insert code to build the event dependencies, which will be accounted for when the events are reordered. In the example of Jacobi1D, relations between `getStripFromLeft`, `getStripFromRight` and `doWork` events can be captured using the overlap and when constructs shown in Figure 4.4. As the structured dagger code runs, a chain of logs preserving the event dependencies are created on the fly. In the new strategy, every event E will have a list of forward and backward dependencies. The backward dependents of E will be those events which must complete before E can start. The forward dependents of E will be the list of events that have E as one of their backward dependents. In the previous example, the event `doWork` has both `getStripFromLeft` and `getStripFromRight` as its backward dependents.

In order to preserve the order between the dependents, an event can be added to the timeline only after all the events that it depends on have been added. To capture this, we define a new term $effRecvTime$

(called effective receive time) recursively as $max(mERT, recvTime)$ where the term $mERT$ is the maximum $effRecvTime$ of all the backward dependents (0 if no backward dependents are present). The $effRecvTime$ is the earliest time in which the event can start such that we maintain the dependency relation between events. The startTime of an event will now be computed as $max(effRecvTime, currTime)$.

The timeline will now be maintained in the non-decreasing order of $effRecvTime$ instead of $recvTime$. The following steps describe this scheme:

```
(1) Calculate the earliest affected event (EAE) in the timeline
(2) Remove all events from the earliest affected event and place them into R
(3) Initialize effRecvTime of those events in R to infinity
(4) Recalculate effRecvTime for all events in R whose backwardDeps are in not R
(5) While R is not empty
(6)     Choose the least effRecvTime event from R
(7)     Reinsert that event into timeline
(8)     Update the effRecvTime of forward dependents of that event
```

When an event gets a new effRecvTime, the EAE in (1) is the earlier of its new position in the timeline and the current position. The steps are performed whenever a correction message arrives. After processing it, the events that have their `startTime` changed will send out correction messages.

In the first version of our BigSim simulator, the high overhead involved in processing the correction messages made the scheme described above very slow. We applied the following optimizations to improve the simulation performance:

- Overwriting timestamps of old messages: When a new correction message arrives, if there is already a correction for the same destination that is not yet processed, the old message's predicted recvTime value is overwritten. The same scheme is used when a correction message arrives earlier than the message itself.

- Using multi-send: Many messages destined to different events but to the same physical processor are sent collectively, using a library provided in Charm++.

- Prioritizing messages based on the recvTime that they carry: This optimization reduces the number of out-of-order messages drastically.

- Lazy processing: Correction messages are processed only periodically. The delay causes many of the correction messages to be overwritten. It also amortizes the cost of restructuring the entire timeline over several corrections.

- Batch processing: Many correction messages are processed collectively. Thus the EAE can be computed for a batch of correction messages and the above scheme is invoked only once for a group of correction messages.

The simulation was still found to be several times slower when timestamp correction was performed. This is caused by the large amount of cascading corrections and numerous out-of-order messages. This occurance of out-of-order messages can be reduced if execution is not allowed to go far ahead of the correction wave. This is because a large difference between them means there are many wrong events to be corrected. Based on these observations a new scheme was developed, as we decribed next.

Approximate GVT scheme

We define the Global Virtual Time (GVT) as the globally smallest timestamp of all pending messages in the system. All the messages (real and correction) currently pending in the system will have a timestamp that is not smaller the current GVT.

We use a heartbeat mechanism to periodically compute the estimated GVT. Every simulated processor of the target machine reports its Simulated Processor Virtual Time (SPVT) by finding the minimum timestamp of all the messages in real and correction message queues and messages sent in its timeline during that interval. The timestamp of a message is the predicted $recvTime$ that it carries. Every real processor computes the Real Processor Virtual Time (RPVT) as the minimum of SPVTs of all simulated processors on it. The minimum of the RPVTs across all processors is the value of the estimated GVT (eGVT) to be used and broadcast in the system. Due to messages in transit that may be arbitrarily delayed, the value of GVT obtained is just an estimate. Therefore, it is not necessary for the new eGVT to be larger than the previous value and the opposite may happen in rare cases. However the trend for the eGVT shall always be increasing, thereby carrying the simulation forward.

After a new eGVT is obtained, to take advantage of the lazy and batch processing optimizations mentioned earlier, we use a time-window to restrain the advance of the program execution beyond eGVT. The correction messages are buffered and processed periodically at every heartbeat. Only those real messages with a timestamp within the window size from eGVT are executed in the heartbeat interval. The time-window advances every time the new eGVT is obtained.

The heartbeat interval is adaptive and we use a heuristic approach to control the advance of the simulation. When the number of messages (real or correction) processed in the last interval is very high, it denotes a high-activity period. The heartbeat interval is then shrunk so that the eGVT can be updated more often to speedup the simulation. Similarly, it is expanded when there is a low-activity period.

4.6 Network Simulation

The current BigSim simulator is suitable for simulating parallel applications whose computation to communication ratio is high, namely, when a simple network model is enough. For applications that require high degree of fidelity to the network model, a much more complicated network simulation is needed.

BigNetSim is a project to incorporate a contention-based network simulation. The first generation of switch-based BigNetSim was developed by Eric Bohm and this author. Praveen Kumar Jagadishprasad and others wrote the second generation of BigNetSim with more detailed network modeling. It is based on the POSE simulation framework [90, 92, 91] developed by Terry Wilmarth.

In BigNetSim, instead of using some preset latency value to determine message transit time, we actually model the message as it passes through a detailed contention-based network model. The power of this approach is that we can model any desired type of network, and plug it into the original timestamp correction simulation. This enables running the application emulation *once*, and reusing the trace logs generated by the emulation to repeatedly analyze the application in a variety of network configurations. Figure 4.5 shows how these components interact.

BigNetSim provides a configurable runtime environment where network parameters, such as buffer size, number of virtual channels, packet size, adaptive vs fixed routing, can be set at run time. Furthermore, the design is modular to support easy extensions by the addition of new topologies, routing algorithms and architectures (see Figure 4.6). BigNetSim currently includes support for Torus, 3D-Mesh, Fat-Tree and Hypercube topologies.

In Figure 4.6, the traffic generator, Network Interface Card (NIC), Switch and Channel are modeled as *posers* (parallel migratable objects for simulating entities) in POSE. They rely exclusively on event messages to communicate with other simulation objects. Messages created on a node are passed to the NIC which

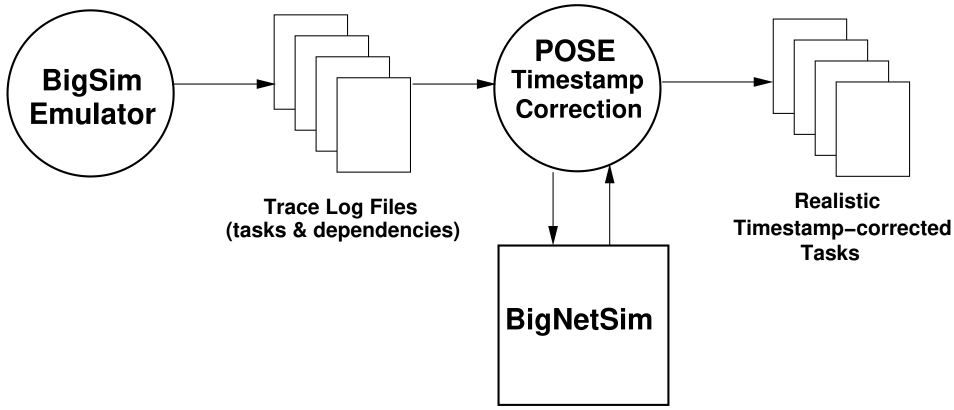


Figure 4.5: Interaction between BigSim, POSE timestamp correction and BigNetSim

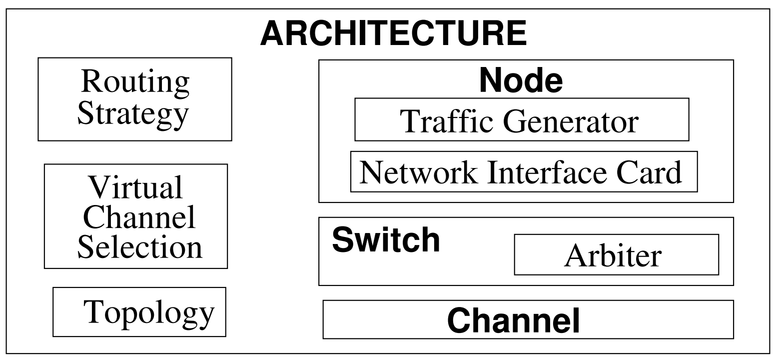


Figure 4.6: BigNetSim conceptual model

packetizes the message and sends the packets to a Channel. The Channel forwards a packet to a Switch. The Switch has an Arbiter, which uses the configured routing logic to select the output Channel to route the packet. Switch models Virtual Channels (VC) and flow control based on feedback from downstream Switches and VC selection strategies. Channels in the path to the destination node receive the packets and pass them up through Switches. Finally the last channel in the path passes it to its NIC, which reassembles packets to form a message and sends it to the node. Several network parameters such as packet-size, channel-bandwidth, channel-delay, switch-buffer-size and a host of NIC delay parameters can be set in a network configuration file, which the simulation reads at runtime.

Simulation of detailed contention-based network models for predicting parallel performance is quite challenging, but we achieved reasonable speedups relative to one-processor parallel time as we present in the next section. POSE's new object model supports for virtualization and adaptive speculative strategies, enabling BigNetSim to perform and scale well despite a seemingly heavy-weight object implementation [92]. Furthermore, based on trace logs generated by BigSim emulator, BigNetSim's network simulation can focus on the task of network simulation, without being disturbed by having to simulate actual computation.

Since network simulation is quite expensive and slow, it is desirable that only the portion of interest in the code be simulated with a detailed network model. As an example, a start-up phase of a parallel application can be skipped without detailed simulation. Log-based postmortem simulation makes this possible. Application developers are allowed to insert “markers” in the parallel code as a potential point to jumpstart the simulation. These markers typically are inserted in synchronization points (such as a barrier) in the application. The markers are then stored in the trace log when the application is emulated. In practice, this scheme has shown to be very effective and efficient.

4.6.1 Parallel Simulator Performance

To evaluate the parallel performance of the simulator itself, we ran the BigSim emulator on 64 real processors with the NAMD program on 2048 simulated nodes configured in a 3D Torus topology with Bluegene/L characteristics. The emulation traces were then used with BigNetSim on a varying number of processors.

We show simulation execution times for BigNetSim with NAMD from 1 to 128 processors in Figure 4.7 and a corresponding speedup plot relative to one processor time in Figure 4.8. The observed data show that BigNetSim scales linearly up to 8 processors , and sub-linearly beyond that.

The results in Figure 4.8 show that BigNetSim has acceptable parallel performance even when applied to performance prediction of a real world application. This is much more challenging than a synthetic benchmark due to complex dependencies among events resulting in relatively limited parallelism in the simulation.

4.7 Validation and Performance Case Studies

We first present results of validation of BigSim on LeMieux [66] at Pittsburgh Supercomputing Center. We then present results of performance prediction and performance analysis of some real world applications for BlueGene/L. Finally, we present the scaling performance of the BigSim itself.

4.7.1 Verification of Jacobi on BG/L

With BigSim, we are now able to study the performance issues of some real world applications on a machine before it is built, specifically BlueGene/L in the following case studies.

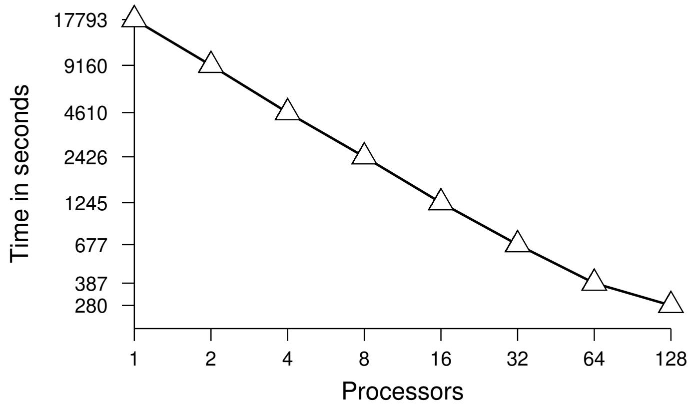


Figure 4.7: BigNetSim execution time with NAMD

To facilitate performance analysis for applications on this machine, *Projections*, a postmortem performance analysis tool associated with CHARM++ has been ported to BigSim. It provides the capabilities of detailed event tracing and interactive graphical analysis.

The Jacobi program written in CHARM++ and Structured Dagger (Section 4.5.2) was used as a case-study to further analyze and verify the simulator. In this simulation, the network model uses a per-hop and per-corner latency of 5ns and 75ns respectively. For experiments with different network configurations, the network latency can be increased by scaling both the per-hop and per-corner latency by the same factor.

The timelines generated by Projections are shown in Figures 4.9 and 4.10 for a selected subset of 64,000 simulated processors. Figure 4.9 was generated without simulation (the program was only emulated), while Figure 4.10 was generated with simulation. The separation between the events in Figure 4.9 is caused by the direct or cascaded effect of the out-of-order delivery of messages. As we can see in Figure 4.10, the timestamps of out-of-order messages were corrected and the gaps disappeared.

We also validated the simulator by comparing the simulation result and the expected behaviors. The results that we obtained are summarized as follows:

- For a valid timestamp correction scheme we expect the same predicted time for a given problem independent of the number of real processors used for simulation. This can be used to verify the simulator. Predicted performance was indeed found to be the same across different runs and the results for Jacobi are shown in Figure 4.11 with different network latency configurations.

- As we increase the network latency we expect the predicted time to remain constant up to a limit

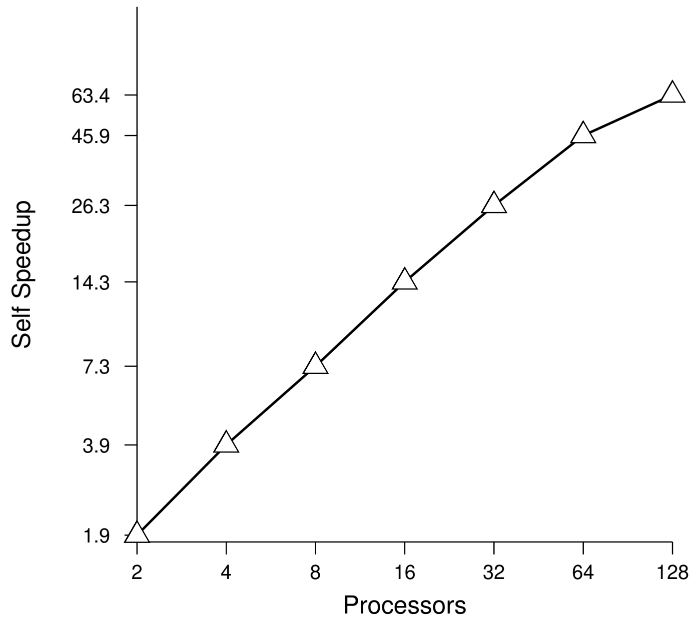


Figure 4.8: BigNetSim speedup with NAMD

and increase thereafter, due to potential overlap of computation and communication. Note that the predicted time was measured as a function of multiplicative factors by which the per-hop and per-corner latencies are increased. The result was as expected and is shown in Figure 4.12.

- The speedup was also measured based on the predicted time for different latency factors as shown in Figure 4.13. For a very low network latency, the speedup was found to be close to linear, and dropped as the latency factor was raised. This is because when the number of simulated processors increases, the work per-processor reduces and the computation can not makeup for communication delay, reducing the speedup.

4.7.2 Validation — AMPI

In order to validate our BigSim, we compared the actual running time of a 7-point stencil program with 3-D decomposition (Jacobi3D) written in MPI with our simulation of it using BigSim. In the program, every chunk of data communicates with its six neighbors in three dimensions. After the Jacobi relaxation computation, the maximum error is calculated via **MPI_Allreduce** among all local errors.

The result is shown in Table 4.1 for a problem with a fixed size in all the runs. The first row in the

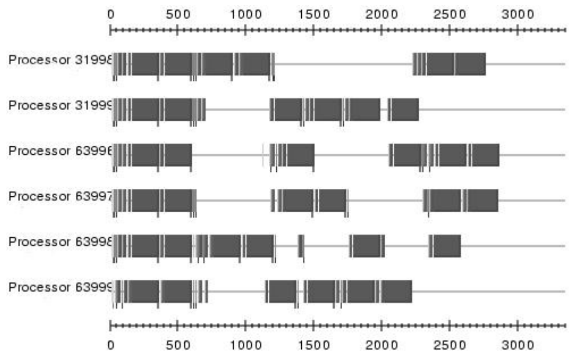


Figure 4.9: Timelines before timestamp correction

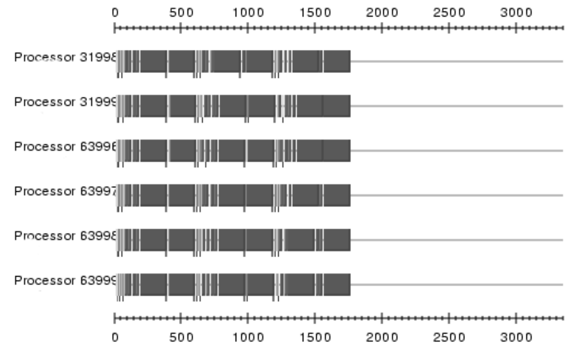


Figure 4.10: Timelines after timestamp correction

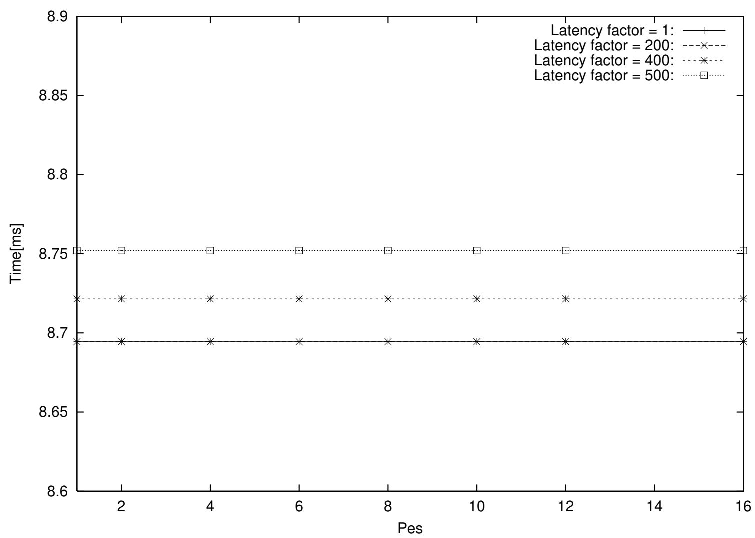


Figure 4.11: Jacobi simulation predictions on different numbers of physical processors used

table shows the running time on 64 to 512 processors; the second row shows the predicted running time when simulating these processors using only 32 real processors. It shows that our simulated execution time generally agrees with the actual execution time to within about 6% although a simple latency based network model is used.

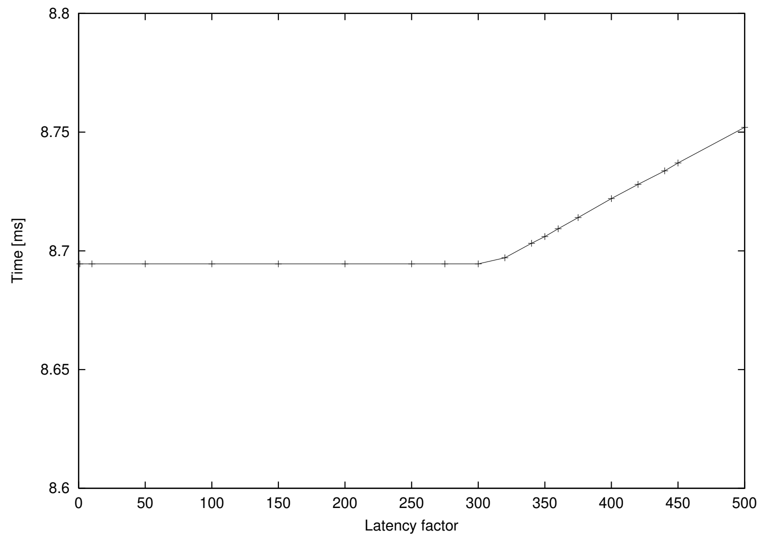


Figure 4.12: Predicted time vs latency multiplier

Processors	64	128	256	512
Actual run time (s)	1.072	0.481	0.259	0.145
predicted time (s)	1.046	0.512	0.270	0.155

Table 4.1: Actual vs. predicted time

4.7.3 Validation — NAMD

We have compared the actual execution time of NAMD with our simulation of it using BigSim on LeMieux[2]. Our validation approach is as follows. We first run NAMD on a number of real processors and record the actual run time. Then we run NAMD on BigSim emulator with a much smaller number of processors simulating the same number of processors used in the original run. This emulation generates log files that we then simulate with BigNetSim running the simple latency-based network model. We record the run time predicted by the simulator and compare with the original run time.

As a NAMD benchmark system we used the Apo-Lipoprotein A1 atom dataset with 92K atoms. The simulation runs for a total of 15 timesteps. A multiple time-stepping scheme with PME (Particle Mesh Ewald) involving a 3D FFT every 4 steps is performed. The result is shown in Table 4.2. The first row shows the actual execution time per timestep of NAMD on 128 to 2250 processors on LeMieux. The second row shows the predicted execution time per timestep using BigNetSim on a Linux cluster, with network parameters based on Quadrics network specifications. It shows that the simulated execution time

[2]LeMieux is 750 Quad Alphaserver ES45 node machine at Pittsburgh Supercomputing Center (PSC)

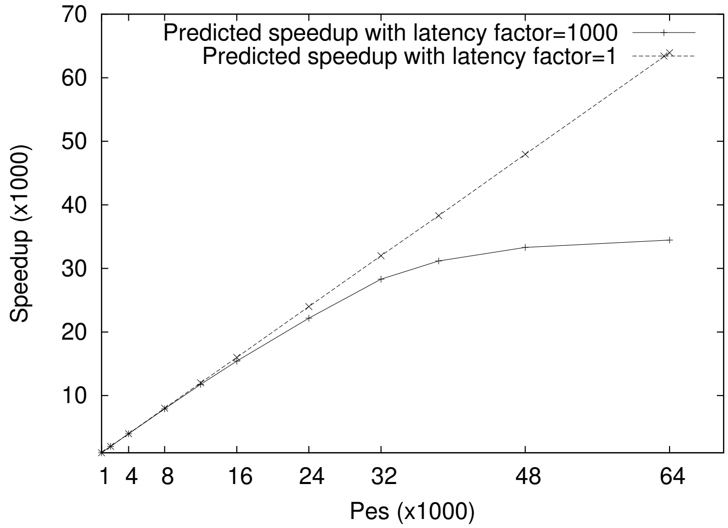


Figure 4.13: Speedup for Jacobi1D

is reasonably close to the actual execution time. The predicted run time is not as accurate when number of real processors grows larger. One possible reason is that we did not model the cache performance and the memory footprint effects in enough detail, hence the predicted time is greater than the actual execution time.

PEs	128	256	512	1024	2250
Actual time	71.5	40.3	23.9	17.6	12.8
Predicted time	75.8	43.6	25.1	20.8	16.13

Table 4.2: Actual vs. predicted time (in *ms*) per timestep for NAMD

NAMD Communication Pattern Analysis with BigNetSim

BigNetSim with detailed network simulation is capable of generating rich information for communication pattern analysis. Network statistics like link utilization and contention obtained from the network simulation can be used to visualize the communication pattern of the application. This helps to identify communication bottlenecks in the applications for performance optimization.

We use NAMD as a case study to illustrate this utility. In NAMD, the atoms are divided spatially into cells roughly the size of a cutoff distance. Local interactions are calculated each timestep between only the nearest neighbor cells. Each simulation timestep starts with multicast communication for cells to send the atom data to the nearest neighbors; the computation begins after that, followed by communication that sends the force result back to the cells. Note that due to the latency tolerance in CHARM++, communication and computation can overlap.

Figure 4.14 shows the average link utilization during the whole simulation in a simulated 128 node 15 time step NAMD simulation. The irregular utilization pattern correlates with bursts of traffic during time step boundaries as shown in Figure 4.15. From the magnified view of a timestep in Figure 4.16, we can see that most communication happens at the beginning and at the end of the timestep as expected.

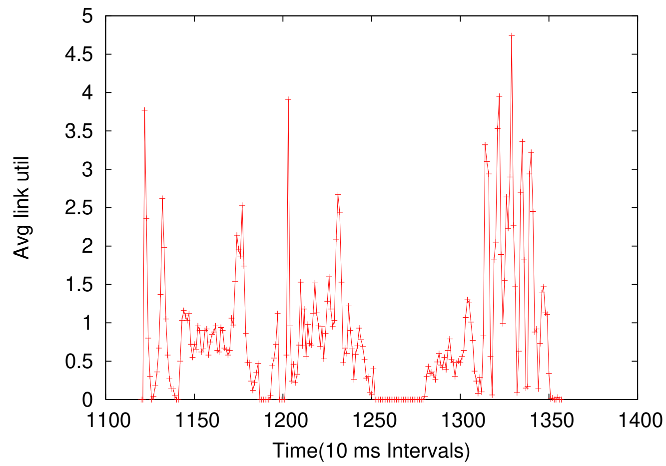


Figure 4.14: Average Link Utilization in NAMD

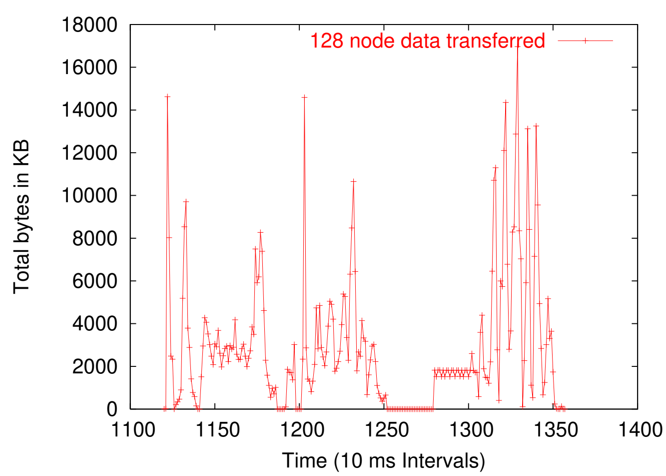


Figure 4.15: Data Transferred (KB) during Full Simulation

The first synchronization is a simple barrier at around the 1200^{th} Interval and the second one is a load balancing step between 1285^{th} and 1315^{th} Interval. The load balancing involves a collection followed by broadcast operation. The traffic results in huge contention as shown in Figure 4.17.

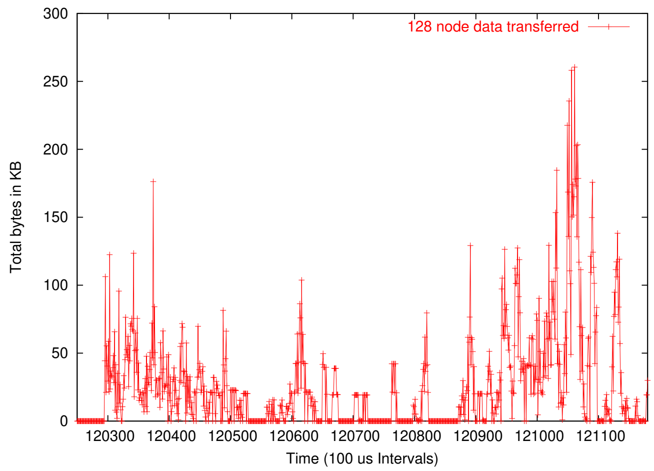


Figure 4.16: Data Transferred (KB) in a Single Time Step

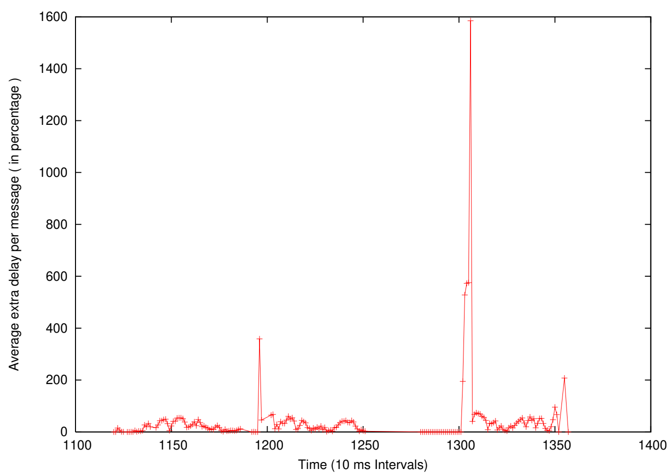


Figure 4.17: Contention encountered by messages

Figure 4.18 shows the number of links where utilization is greater than 30 percent in an Interval. We can clearly see that many links during load balancing have utilization greater than 30 percent. This is significant as a typical message is transferred in a few microseconds and it takes many messages to be transferred in 10 milliseconds to have a high link utilization. Blue Gene/L has a separate tree network for doing collective operations, which is not modeled in the current network simulator. We believe that the tree network will help alleviate the network contention of such broadcasts.

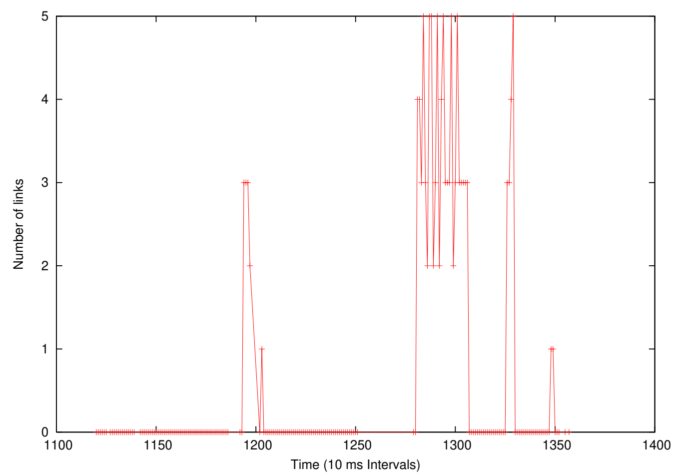


Figure 4.18: Number of links with utilization greater than 30 percent

4.7.4 Performance of the Simulation

We also measured the performance of the simulator itself using LeanMD as a sample application. We demonstrate the scalability of the parallel simulator in Figure 4.19. The simulation was found to scale reasonably over hundreds of processors. The efficiency of the simulation depends on the number of correction messages sent. In one simulation, correction and real messages sent were compared for different simulated processors as shown in Table 4.3. The low ratio of correction messages to real messages was encouraging, which leads to considerable performance improvement of the simulation.

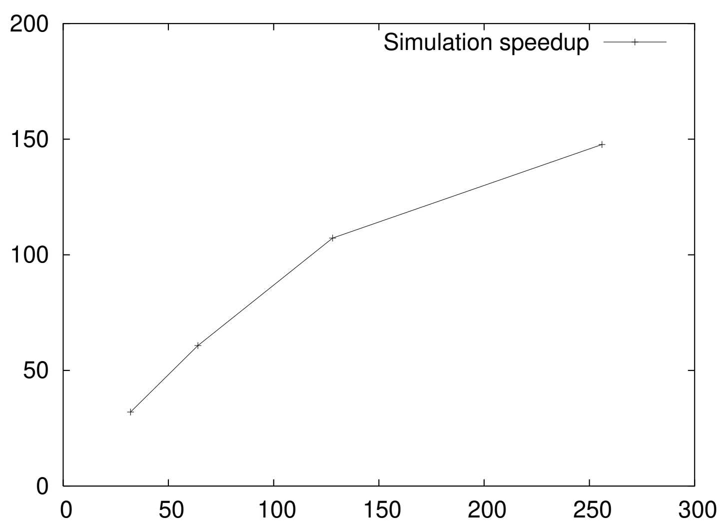


Figure 4.19: Simulation Speedup

Processors	8k	16K	32k	64k
Real Msgs	20.04M	20.18M	20.42M	20.93M
Correction Msgs	357351	305487	126629	59762

Table 4.3: Proportion of correction messages

Chapter 5

Million-way Parallelization

Parallel algorithms developed for conventional parallel machines are not necessarily appropriate or efficient for peta-scale machines. Parallel algorithms for such class of peta-scale machines must handle low bisection bandwidth and relatively low memory-to-processor ratio. They must exploit the availability of dedicated communication threads and the existence of multiple parallel communication links. In order for load balancing to work effectively, parallel applications need to create a high degree of parallelism such as migratable objects in Charm++ and AMPI.

In our work, we have developed and evaluated two parallel frameworks and their applications written in Charm++ and AMPI for peta-scale class machines. They are Molecular Dynamics (MD) and FEM framework.

5.1 Molecular Dynamics Simulation - LeanMD

The molecular dynamics simulation of biomolecules is one of the planned applications for BlueGene/L. It is a challenging application to parallelize. A microsecond simulation includes about a billion timesteps. Each timestep involves a relatively small amount of computation that must be effectively parallelized. We have developed **NAMD**, a Gordon Bell award winning parallel molecular dynamics application that is written in CHARM++. Although it has been shown to scale to 3000 processors [54], it is not ready for extreme-scale parallel machines due to the relatively limited parallelism exploited in the application.

In NAMD, the atoms in the simulation are divided spatially into cells roughly the size of the cutoff distance. Local interactions are calculated for each timestep between the nearest neighbor cells ("one-away" interactions), as illustrated in Figure 5.1. This ensures that all atoms within the cutoff radius are

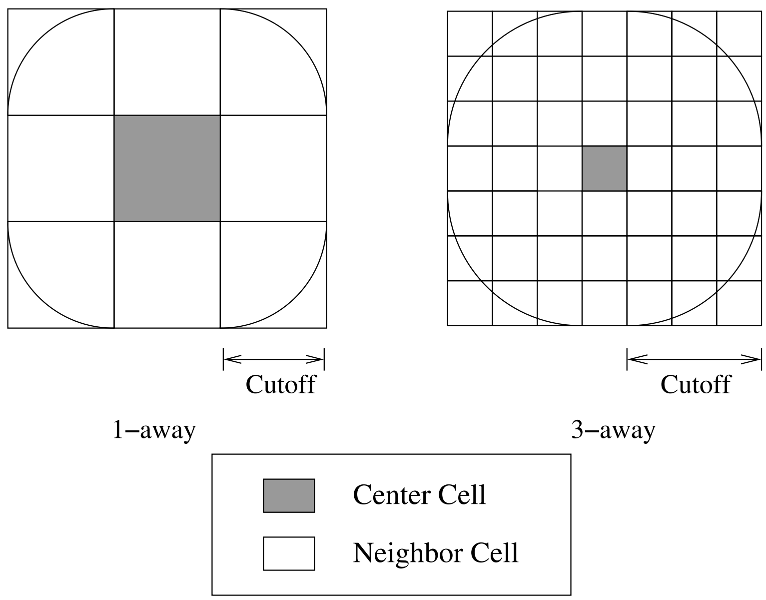
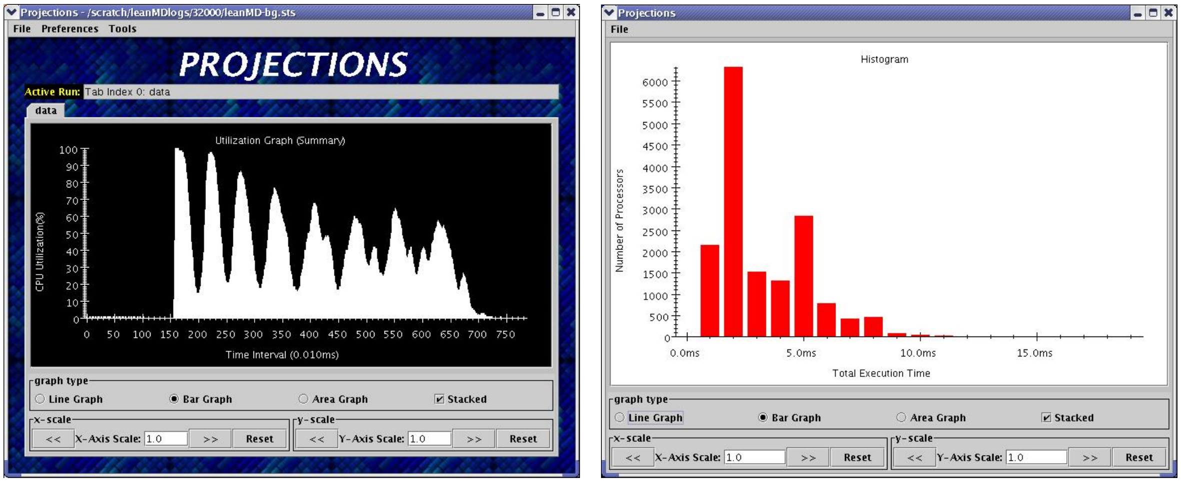


Figure 5.1: 1-away and 3-away cut-off distance

considered. However, this strategy produces a division that is too coarse grained for planned machines such as BlueGene/L. For example, with a cutoff radius of 15 Å, a 150 x 150 x 150 Å simulation space would involve only 1,000 cells and 13,000 cell-to-cell interactions [1] to calculate as in one-away scheme. Considering that the BlueGene/L machine is approximately 64,000 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, **LeanMD** was developed as an experimental code. 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 5.1. 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 can be easily distributed across the 64,000 nodes of BlueGene/L.



(a) Average utilization per interval for LeanMD on 32,000 processors

(b) Distribution of processors based on load in ms

Figure 5.2: LeanMD Projections Views

Experimental Results

We have run LeanMD in our simulator on PSC LeMieux using the ER-GRE atom benchmark, simulating a BlueGene/L machine of size varying from 1K to 64K nodes. The simulation data can be used to carry out a detailed performance analysis using the Projections tool. Figure 5.2(a) shows how the average processor utilization varies with time for a simulation on 32k simulated processors. The utilization stabilizes at about 50%, but rises and falls within each timestep. To further understand the scalability of LeanMD, we ran LeanMD with simulations from 1K to 64K simulated processors. The first row in Table 5.1 is the predicted speedup, normalized based on the 1000-processor time. The speedup starts to degradate significantly at 16k simulated processors. This could be due to communication latencies, critical paths or load imbalance. To understand the degradation of the speedup we used the performance data to calculate the CPU load on each individual processor. Figure 5.2(b) shows a histogram of this data in the case of 16k simulated processors. Although about 6000 out of 16000 processors have a load of about 2ms, a few are seen to have a load as high as 11ms. This suggests that load balance is a major performance issue. To understand what portion of performance loss is explained by load imbalance alone, we calculate the estimated speedup ($P \times \frac{avgLoad}{maxLoad}$) based on load imbalance loss alone (second row in Table 5.1) and compare it with simulated speedup. The

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

closeness of both numbers confirms that load imbalance is the primary cause of performance loss. Only at 64K processors do the numbers deviate, indicating influence of other factors such as communication overhead or critical paths. Such detailed performance analysis is possible because of the rich performance trace data produced by the simulator.

Processors	2000	4000	8000	16000	32000	64000
Simulated Speedup	1845	3384	6015	8658	14178	18180
Estimated Speedup	1865	3412	6242	8635	13916	19936

Table 5.1: Simulated speedup vs. estimated speedup (based on load imbalance alone), normalized based on 1000-processor case

5.2 Finite Element Methods Simulation

The Finite Element Method (FEM) is a popular technique often used in the study of fracture and structural mechanics. We have developed a parallel framework[18], called the CHARM++ FEM Framework, to make it easy to parallelize a serial FEM code. It is written in AMPI to take advantage of the migratable objects and the idea of processor virtualization. The framework handles the finite element mesh that discretizes the problem domain, partitions the mesh for parallel execution, and provides an easy way to use communication primitives defined on the mesh.

It is very difficult for an FEM application to generate million-way parallelism due to several practical constraints and bottlenecks in the existing tools, which we discuss next. We have improved tools like METIS (a sequential mesh partitioning library) towards very large scale mesh partitioning, and we are now able to partition a mesh into 20,000 pieces. We plan to port tools like ParMetis — a parallel mesh partitioning library — on AMPI to allow even bigger partitioning.

Bottlenecks on Large Machines

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

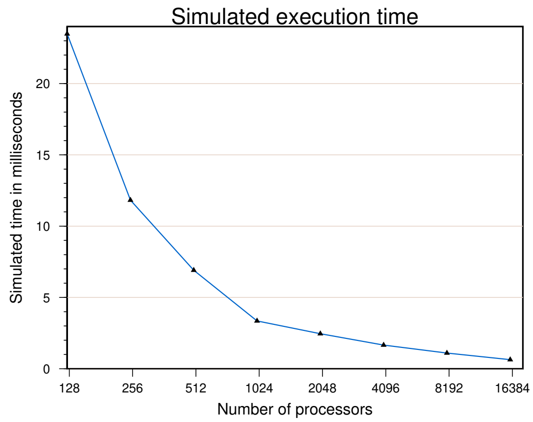
Our runs with BigSim also exposed a number of unexpected bottlenecks and limitations to scalability. For example, we found several "bugs" in the METIS library that prevented it from partitioning a mesh into more than a few thousand pieces. One bug in METIS was the loss of floating-point precision when using single precision in load vector representation for partitions. As the recursion of the partitioning goes deeper, the loss of floating point precision becomes so overwhelmingly big that the sum of the load vector is not constant any more, leading to a crash in the library. We found that promoting values to double precision fixed this problem. Another bug was related to the problem of integer overflow. In METIS, to partition a mesh into K pieces, an integer array of size K^2 is allocated. However, since the array index type was defined as "integer", a 50,000 piece output partition could overflow a 32-bit integer representation. Promoting the index type from "integer" to "unsigned integer" or "long" fixed the problem.

Even with these fixes, METIS uses a large amount of memory due to the K^2 memory allocation, so it consumes memory proportional to the number of output pieces, not the total size of the mesh. Thus, a 4GB machine ran out of memory when partitioning a relatively small (5M elements) mesh into more than 16K pieces. We believe ParMetis (parallel Metis) will solve this problem, and we are investigating the integration of parMetis into the FEM framework and AMPI.

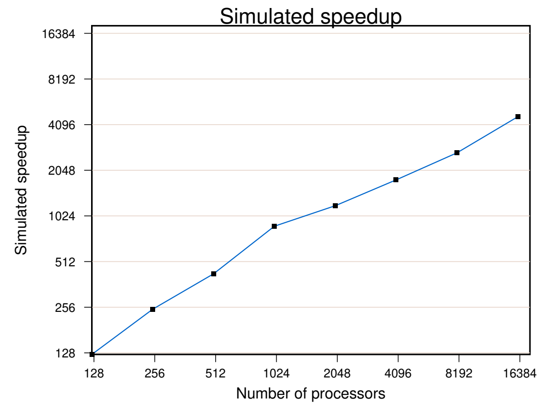
Similarly, even though our MPI implementation, AMPI, was designed to be scalable, while trying to simulate very large machines we discovered that our implementation used total memory proportional to P^2 for P processors. The culprit was a simple linear message ordering table kept by each processor, as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

Experimental Results

We studied the performance of a CHARM++ FEM Framework program, which performs a simple 2D structural simulation on an unstructured triangle mesh. We chose a relatively small problem with a 5 million element mesh, to stress efficiency issues. Because our 2D elements take a little under a microsecond of CPU time per timestep, this corresponds to less than 5 seconds of serial work per timestep.



(a) Simulated execution time



(b) Simulated speedup

Figure 5.3: FEM Framework Performance on a 5 Million Element Mesh

Figure 5.3(a) shows the predicted execution time per step, simulating 125 to 16,000 processors using only 32 LeMieux processors. The time per step is 23.5 milliseconds for 125 processors and drops to 640 microseconds on 16,000 processors. Figure 5.3(b) is the corresponding speedup, normalized based on the 125-processor time. It shows that the program can scale well to at least several thousands of processors.

Beyond several thousand processors, when the simulated time per step drops below a few milliseconds, the parallel efficiency begins to drop. Sub-millisecond cycle times are indeed extremely challenging even on today's small machines, and we continue to seek methods to improve this performance on larger machines.

We also demonstrate the benefits of processor virtualization in CHARM++ for the same FEM program. We use different numbers of MPI virtual processors, each with a separate chunk of the problem mesh, on each simulated processor. Virtualization allows dynamic overlap of computation and communication, and can improve cache utilization because each virtual processor's data is small.

The predicted performance for various degrees of virtualization is illustrated in Figure 5.4. The problem size in this test is still the same—a 5 million element mesh, and the simulated machine size is fixed at 2000 [2]. Even a low degree of virtualization significantly improves performance by allowing computation and communication to be overlapped; higher degrees of virtualization provide little benefit, and eventually the overhead of additional virtual processors only slows the program down.

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

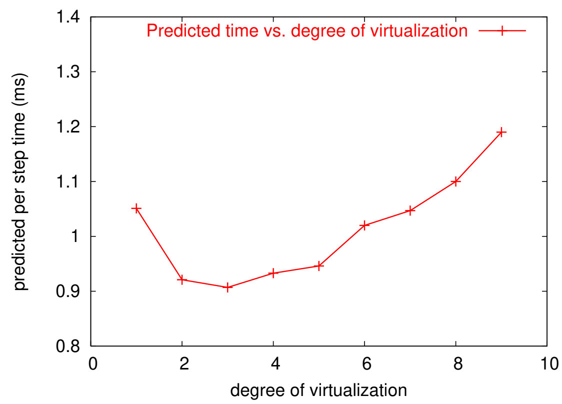


Figure 5.4: Predicted execution time for a 5 million element mesh vs. degree of virtualization

Chapter 6

Dynamic Load Balancing Framework

Load balancing is a technique that performs the task of distributing computation and communication activities evenly across processors of a parallel machine so that no single processor is overloaded. Load balancing is one of the key factors for achieving high performance on large parallel machines when solving highly dynamic and irregular problems.

The load balancing problem involves making decisions on placing newly created computational tasks on processors, or migrating existing tasks among processors in favor of load balance. The load balancing problem is NP-complete. However, many combinatorial algorithms have been developed that find a reasonably good approximate solution (Section 6.3).

Load balancing problem is also a challenge to software design. Most existing dynamic load balancing often is added directly to applications through a built-in load balancing algorithm implemented specific to the application. This approach has a number of disadvantages. Because the load balancing is directly implemented in an application, the application developer can not compare the algorithm to other strategies easily to evaluate their effectiveness. Since the load balancing is too tightly coupled with the application, the resulting implementation cannot be used in other applications in a straightforward way. The application developer may not have the expertise or interest to optimize load balancing algorithm since their primary interest is the application itself.

In this chapter, we discuss a complete and practical automatic dynamic load balancing methodology based on run-time instrumentation. It is unique in providing dynamic load balancing to a wide range of dynamic, unstructured and/or adaptive applications. It delivers this support without creating burdens to application developers.

6.1 The Load Balancing Problem

The goal of load balancing can be stated as follows [1]:

> *Given a collection of tasks that involve computation and communication and a set of computers connected in a certain topology, find a mapping of tasks to computers such that each computer has an approximately equal amount of computation and communication between processors is minimized.*

Tasks can be calls to the subroutines, migratable objects and threads, etc. A mapping that balances the workload of tasks on processors potentially leads to an increase in the overall efficiency of parallel execution. However, a naive balancing of the load is not guaranteed to improve the parallel efficiency.

6.1.1 Basic Definitions

In this thesis, we study the load balancing problem in the context of Charm++ and parallel objects. We first gives some basic definitions used in this thesis.

- **Processor Graph**

  Processors of a machine may be organized in a network topology. The network topology can be represented as an undirected graph $G_p = (V_p, E_p)$ on $p\ (= |V_p|)$ vertices. Each vertex in V_p represents a processor, and the edges in E_p represent a direct link in the network.

- **Task Graph** The parallel application is represented as a weighted undirected graph $G_o = (V_o, E_o)$. The vertices in V_o represent parallel objects (or groups of objects) and the edges in E_o represent direct communication between the parallel objects (or groups of objects). Each vertex $v \in V_o$ has a weight w. The weight on a vertex denotes the amount of *computation* that the objects in the vertex represent. Similarly, each edge $e_{ab} = (v_a, v_b) \in E_o$ has a weight c_{ab}. The weight c_{ab} represents the amount of *communication* in bytes between the parallel objects represented by v_a and v_b. The edges are undirected, which means that we are not considering the direction of communication in our algorithm.

[1] This definition is similar to that in [88] but is more comprehensive.

- **Task Mapping** The task-mapping is represented by a map function:

$$P : V_o \longrightarrow V_p$$

If the parallel objects represented by the vertex $o \in V_o$ of the task-graph are placed on processor p, then $P(o) = p$.

After mapping, each vertex $p \in V_p$ has a weight W_p representing the total load on processor p, which is the sum of all objects on this processor:

$$W_p = \sum_{o \in O} w_o, \qquad O = \{o | P(o) = p\}$$

To consider communication overhead in the weight W_p for a processor $p \in V_p$, we add all non-local communication cost representing by the sum of weights of all outgoing edges:

$$W_p = \sum_{o \in O} (w_o + \sum_{e_{ob} \in E_o \wedge P(b) != p} c_{ob}), \qquad O = \{o | P(o) = p\}$$

It is worthwhile to distinguish between *problem decomposition* and *task mapping*. Problem decomposition is a part of the parallel program design process; it tries to decompose an application into several modules to run on separate processors. The result of problem decomposition is a collection of tasks that solve the problem in parallel. These tasks can be independent or with connections through communication. Task mapping is a separate process that maps the above resultant tasks to processors. In Charm++'s virtualization approach, an application should ideally be decomposed into a number of tasks that is much greater than the number of processors to allow load balancing schemes to perform task remapping.

6.2 Application Load Imbalance

An increasing number of emerging parallel applications exhibit dynamic and/or irregular computational structure. These applications are likely to experience load imbalance when running on parallel computing platforms.

Application imbalance is a major contributor to inefficiency in parallel programs, especially on a dedicated parallel machine free of external interference. With load imbalance, a parallel application can only execute at the speed of the most heavily loaded processor.

Given an accurate model of computation and communication costs, it may be possible to compute all possible work distributions for a given algorithm and select the one that gives best load balance. In practice however, this NP-hard problem is often too complex to solve exactly. Furthermore, such an accurate model for an application is difficult to get, especially when the computation and communication patterns change over time. For this reason, static load balancing is not the focus of this thesis.

Dynamic load balancing tries to solve the load balance problem at run-time according to the most up-to-date load situation. It can efficiently solve problems of irregular structure with instrumented load estimates on multiprocessor systems. Rebalancing is required when the load balance in the application changes over time.

6.3 Related Work and Load Balancing Contexts

Both the dynamic behavior of applications and machine characteristics influence the design of load balancing strategies. In this section, we discuss the three categories of the application characteristics, describe the machine characteristics that influence the load balancing algorithms, and summarize task migration mechanisms.

6.3.1 Non-iterative Applications

Non-iterative applications have no temporal correlation and lack predictability. As an example, consider a state-space searching application used for problem solving. A state space is the set of discrete configurations (i.e. states) that might be encountered while trying to solve a problem. A state space search entails systematically exploring these states in an effort to find a solution to the problem. Sometimes, any solution will be acceptable (e.g. to find a k-coloring for a graph). But frequently, a solution that can be reached with the least possible cost (an optimal solution) is desired. In a parallel algorithm, the problem can be decomposed into subtasks, with each subtask responsible for a subset of state space searching. These subtasks are short-lived tasks, but may spawn new subtasks. Due to the fact that new subtasks are generated based on the processing of an existing task, the load balancer has to make decisions solely based on new tasks and with limited information about the global load distribution. Since there is no predictability to the computation, it is difficult to obtain a global load distribution information for load balancing algorithm.

This load balancing problem is also known as *task scheduling problem* [46, 80] in the literature. A task scheduling problem can be stated as follows: given a set of tasks and a number of processors that can concurrently execute the tasks, find an efficient way to schedule the execution of these tasks to obtain better task response time, throughput and resource utilization, while keeping the scheduling overhead to a minimum.

The task scheduling problem has been extensively studied in the literature. Essentially, scheduling solutions can be classified into two main categories. The first category is *static scheduling techniques*, which perform the scheduling of communicating tasks before run time. This scheduling scheme often requires *a priori* knowledge of the tasks being scheduled. Graph theoretic models such as network flow [19] and heuristics [36] are commonly used to make the task assignment.

The second class of scheduling does not assume *a priori* knowledge about the tasks scheduled and make the scheduling decision at run time. These scheduling schemes are known as *dynamic load balancing strategies* [61, 14]. This class of scheduling schemes often assumes a continuous flow of tasks which may be processing modules that are either independent or dependent on each other.

In Charm++, the task scheduling problem is treated as *seed load balancing*. Tasks are created as parallel objects, which are treated as "seeds" for which an initial processor placement must be determined by the load balancing system. The first Charm++ seed load balancer was developed in the late 80's. In [46, 80], the authors proposed and implemented an algorithm in Chare-Kernel (early Charm++) known as Adaptive Contracting Within Neighborhood (ACWN). In [78], Saletore presented a neighborhood averaging scheme for parallel processing of medium-Grain tasks. In ACWN, work always travels to topologically adjacent neighbors with the least load, but only if the difference in loads is more than a predefined threshold. In addition, ACWN does saturation control by classifying the system as being either lightly, moderately or heavily loaded. Sinha et al. [82] described an efficient prioritized load balancing strategy for prioritized execution of tasks.

6.3.2 Iterative Applications with Predictability

The second category of applications includes those that are *iterative* with *predictability*. In fact, many scientific applications, especially physical simulations, are iterative in nature. The computation lasts for a long time and may contain a series of time-steps and/or iterative solvers that run to convergence. For many such problems, computational loads and communication patterns between tasks tend to persist over time. Thus, the recent load history is a good predictor for the near future. This heuristic is called the "*principle of persistence*" [52]. For these applications, a measurement-based load balancing method is useful.

In the context of CHARM++ and migratable objects, depending on the degree of the "persistence", the following instrumentation schemes can be considered in the run-time system, in increasing order of complexity:

1. Object load only: only CPU load is measured and no communication pattern is recorded in this method;

2. Object-communication graph: both the CPU load and the communication pattern are instrumented, so that one can build an object-communication graph;

3. Multiple-phases: computation and communication only tend to be persistent in multiple phases. Statistics collection needs to be done in these phases separately;

4. Task dependence graph: when communication occurs in a specific order that is describable by a task dependence graph or a critical path.

Each of these scenarios is useful in specific applications, and needs a separate set of strategies. We will focus on first three scenarios in CHARM++ in Chapter 7.

6.3.3 Iterative Applications without Predictability

The third category of applications includes those that are *iterative* but with *no predictability*. These applications may involve iterative computations with correlation between iterations/steps, but either the correlation is not strong or the machine environment is not predictable, leading to abrupt changes in load. Such unpredictability may occur when there is noise due to operating system interference on small time-steps, or on time-shared desktop machines. For applications in such scenarios, it is difficult to balance the load. One possible solution is to apply a mixed strategy that combines both the measurement-based and the seed load balancing strategies, such as a *sub-step load balancing* strategy described in Section 7.2.3.

6.3.4 Partitioning Algorithms

Load balancing problem can be essentially viewed as a partitioning problem that assigns a set of computation tasks on target processors. Partitioning algorithms can range from simple algorithms that involve communication-oblivious partitioning to sophisticated algorithms that take communication into account.

Communication-oblivious partitioning

The problem of assigning a set of n jobs (each with some arbitrary size) on p processors, so as to minimize the maximum load (makespan) on the processors is called the *Makespan minimization problem* and is well studied. It is an NP-hard optimization problem [67]. Hochbaum and Shmoys give a polynomial approximation scheme for uniform machines [40]. T.D.Braun et.al. [20] study a variety of mapping heuristics for heterogeneous machines. Min-min heuristic assigns the smallest overall job to its biggest machine, Max-min assigns the largest overall job to its best machine. Search techniques like genetic algorithms (GA), simulated annealing, and A* are also considered. It is observed that search techniques like GA give the best result but are quite slow [20]. For homogeneous systems, a simple greedy algorithm where jobs are assigned in decreasing order of sizes to the current least loaded machine performs well in practice. Though the algorithm can guarantee only a 2-approximation, it gives much better results with jobs being small (relative to average load on processors). A refinement pass, after such a greedy assignment, which looks at overloaded processors and swaps work with underloaded processors, can further improve performance.

Communication-aware partitioning

In the context of parallel programs, a model where tasks *communicate* better reflects reality. While it is desirable to reduce maximum CPU load, performance is also effected by communication requirements. Thus applications with high connectivity require more sophisticated load balancing strategy that optimizes application communication. Communication in parallel applications can be considered implicitly (such as in geometric partitioning) or explicitly (such as in graph partitioning).

Geometric partitioning is an old, conceptually simple techniques for quickly and inexpensively generating decompositions that preserve locality. Instead of take communication graph into account explicitly, it uses only the geometric coordinates of objects to assign regions of space to processors so that the weight of objects in each region is equal. This method is particularly efficient for applications that geometric proximity of objects are more important than their graph connectivity. Orthogonal recursive bisection (ORB) method [16, 81] and space-filling curve (SFC) [87, 71] partitioning are examples of such strategies.

Recursive bisection [16, 81] computes a cutting plane that divides a dimensional space into two subregions, each with half of the total computation work (Figure 6.1). This cutting procedure is applied recursively to each subregion until the number of subregions equals the number of partitions desired.

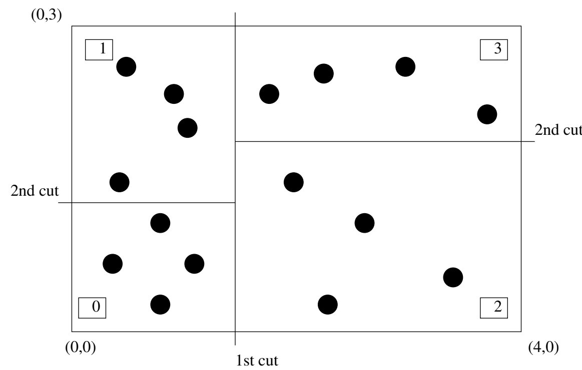


Figure 6.1: Recursive bisection

A space-filling curve (SFC) maps n-dimensional space to one dimension [77]. In SFC partitioning, an object's coordinates are converted to a key representing the object's position along a SFC through the physical domain. Sorting the keys gives a linear ordering of the objects (Figure 6.2). This ordering is cut into pieces with appropriate weight which are then assigned to processors.

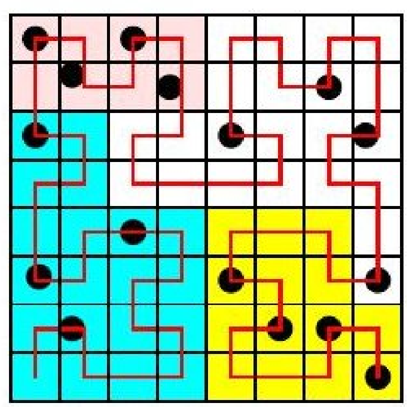


Figure 6.2: Space-filling curve

Geometic partitioning methods do not explicitly control communication, therefore they are effective only for applications that the geometic locality is important. For the same reason, the load balancing result may induce higher communication costs than graph-based algorithms that explicitly control communication for some applications. However, due to their simplicity, they generally run faster and are easier to implement. CHARM++ load balancing framework implements the orthogonal recursive bisection partitioner (OrbLB) with enhancement of adaptivity. Taking processor background load into account, OrbLB migrates parallel objects away from processors with high background load to tolerate O.S. interference.

Graph partitioning is a useful abstraction for load balancing applications with intensive communication. The main idea is to represent the computational application as a weighted graph. The nodes or vertices in the graph correspond to objects. Each object may have a weight that normally represents the amount of computation. The edges or arcs in the graph usually correspond to communication costs. In graph partitioning, the problem is to find a partitioning of the graph (that is, each vertex is assigned to one out of k possible sets called partitions) that minimizes the cut size (weight) subject to the partitions having approximately equal size (weight). All the problems described above are NP-hard so no efficient exact algorithm is known.

Stramm and Berman [86] formulate cost function that model processor loads and communication. They use these cost functions to guide local neighborhood search and simulated annealing search over the possible configuration space. It is observed that a good starting configuration is needed for these searches to lead to a good result. Mansour et.al. [70] present a graph contraction algorithm where highly communicating neighbors are repeatedly merged. The Kernighan-Lin algorithm [63] starts with an initial mapping and refines it by swapping pairs of objects across group boundaries, so that cross-edges are reduced.

METIS is a set of programs that implement various graph partitioning algorithms that are based on the Recursive multilevel k-way partitioning algorithm [56]. To partition an unstructured graph, the multilevel partitioning algorithms first reduce the size of the graph by collapsing vertices and edges, partition the smaller graph, and then uncoarsen it to construct a partition for the original graph. METIS uses novel approaches to successively reduce the size of the graph as well as to further refine the partition during the uncoarsening phase. During coarsening, METIS employs algorithms that make it easier to find a high-quality partition at the coarsest graph. During refinement, METIS focuses primarily on the portion of the graph that is close to the partition boundary. These highly tuned algorithms allow METIS to quickly produce high-quality partitions for a large variety of graphs.

Centralized v.s. Distributed Load Balancing Strategy

The size of a machine can also influence the way in which load balancing is carried out. It can be centralized for relatively small machines or fully distributed for large machines.

Most of the dynamic load balancing strategies can be classified as centralized [26, 72] or fully distributed methods [28, 46, 39]. In centralized strategies, a dedicated "central" processor gathers global information about the state of the entire machine and uses it to make global load balancing decisions. On the other hand, in a fully distributed strategy, each processor exchanges state information with other processors in its neighborhood. Fully distributed strategies have been proposed, typically in the context of non-iterative tasks. These include neighborhood averaging scheme (ACWN) [46, 80, 83], and a set of strategies proposed in [89]. In [89], several distributed and hierarchical load balancing strategies are studied, such as Sender/Receiver Initiated Diffusion (SID/RID), Gradient Model (GM) and Dimension Exchange Method (DEM) and a Hierarchical Balancing Method (HBM). Other hierarchical strategies are explored in Furuichi et al. [38] and Sinha et al. [82]. These are discussed in more detail in Section 8.5.

6.3.5 Task Migration

Load balance can be achieved by migrating "work" from one processor to another. In one popular approach, migrating groups of communicating processes or threads, i.e. *process migration* was studied as a means for supporting load balancing on workstation clusters [35, 68] as well as fault tolerance.

Sprite [34] was an early effort to support transparent process migration and file access across a network of clusters. In Sprite, each process is initially created on a home processor. Later, this process may migrate to less-loaded processors. However, when the process performs I/O, it needs to migrate back to its home processor.

MOSIX [12, 11] is a management system that allows a Linux cluster or an organizational grid to perform like a single computer with multiple processors. It is particularly suitable to run intensive computing and applications with moderate amounts of I/O. In MOSIX, processes are transparently migrated. Like Sprite, processes are first created on a home processor, and then later migrate to other processors when needed. Similarly, when performing I/O, the process has to migrate back to its home processor. Applications just need to "fork processes and forget". The run-time system monitors and initiates process migration to improve the overall performance by moving processes from slower to faster nodes or by migrating processes from nodes that run out of memory.

Process migration for load balancing has many limitations. First, it adds to the complexity of the runtime system and is often difficult to port. Secondly, as the image of a Unix process is large, it requires significant communication and a consequent delay before a process can be migrated. Thirdly, since a whole process as a migration unit is relatively big, moving a process to another processor may cause new load imbalance problems on the destination processor. Thus load balancing based on process migration is not suitable for massively parallel machines.

Data migration [31, 14, 49] implicitly migrates data (and hence computation) among processors to achieve load balance. It is more portable and simpler to implement. However, data migration often requires knowledge about the application and shifts the entire burden to the application programmer.

PREMA [14, 13] implements *Schedulable Objects* similar to CHARM++ migratable objects. In order to migrate a schedulable object, three additional callback routines are needed in the application: one to pack the schedulable object into a contiguous buffer to send as a message, one to unpack the schedulable object from a buffer, and one to return the size of the schedulable object in bytes for memory allocation.

Zoltan [31, 32] is a library that is a collection of data management services for parallel, unstructured, adaptive, and dynamic applications. It provides three classes of parallel partitioning algorithms: geometry bisection, space-filling curves, and the graph partitioning. It simplifies the load balancing, data movement and unstructured-communication, that arise in dynamic applications such as adaptive finite-element methods, particle methods, and crash simulations.

6.3.6 Our Approach

Software design is an important part of dynamic load balancing research. One of the goal of this research in this thesis is to present an enhanced software framework which can be used to quickly develop and easily evaluate customized load balancing algorithms to address the different needs of load balancing. In particular, we wish to distinguish our research using the following criteria:

- Support for data migration. Migrating data has advantages over migrating processes or threads which adds complexity to the runtime system.
- Application independent load balancing. Load balancing framework does not require application knowledge to make load balancing decisions.

- Take communication into account explicitly rather than implicitly for example using domain specific knowledge. Communication pattern including multicast and communication volume are directly recorded into load balancing database for making load balancing decisions. Section 7.3 will demonstrate the advantage of this approach.

- Automatic load estimation. Instead of relying on applications to predict work load of computation, our framework automatically instrument object computation and communication pattern without user intervention.

- Adaptive to execution environment. Take background load and non-migratable load into account.

Table 6.1 shows the comparison of CHARM++ load balancing framework to several other software systems that support dynamic load balancing. DRAMA [15] is designed specifically to support finite element applications. This specialization enables DRAMA to provide an application "independent" load balancing using its built-in cost functions for the category of its applications. Zoltan [31, 32] does not make assumptions about applications' data, and is designed to be general purpose load balancing tools. However, it relies on application developers to provide cost function and communication graph. A recent system PREMA [13, 14] supports very similar idea of migratable objects, however, its load balancing support primarily focuses on task scheduling problem as in non-iterative applications.

System Name	data migration	application independent	explicit comm.	automatic cost	adaptive
DRAMA	yes	yes	no	yes	no
Zoltan	yes	yes	no	no	no
PREMA	yes	no	no	no	no
CHARM++	yes	yes	yes	yes	yes

Table 6.1: Software systems that support dynamic load balancing

6.4 Practical Load Balancing Methodology

Typically, a practical and complete load balancing solution involves four distinct steps: (1) Load evaluation; (2) Load balancing initiation; (3) Load balance decision making and (4) Object migration [2]. The following sections elaborate each step of load balancing with various design considerations.

[2]A similar sequence is described in [89]

6.4.1 Load Evaluation

A load evaluation module provides estimation of the load on each processor. It provides the load information as a "database" for the load balancing process to (a) determine if a load imbalance occurs and (b) to compute a new mapping of tasks by calculating how much work to migrate. Thus, the accuracy and completeness of the information in the database directly affect the quality of the load balancing output.

Typically, such a database contains two categories of data. One type of data is processor-level information. It includes total CPU load, idle time and background load, etc. The other type of data is object-level information. This contains much more detailed information for each parallel object, including execution time and communication pattern and overhead.

Communication pattern is an important information for load balancers to make migration decisions. If one has to move an object from an overloaded processor A to a underloaded processor B, it is beneficial to move that object which communicates most with objects already on B. Two types of communication among objects must be taken into account: point-to-point communications and collective communications, such as multicast. When a load balancer is considering communication in making object-to-processor assignment, it estimates the cost of the communication. Point-to-point communication overhead between a pair of objects can be represented by the volume of communication (b in bytes) and the frequency of the communication (n as the number of messages per time unit). Given the per message CPU overhead α and per byte CPU overhead β, the total communication overhead between the pair of objects imposed on the processor can be estimated by:

$$\alpha * n + \beta * b$$

where b is the total volume of all n messages.

Processor and object load evaluation can be done in several ways. One way, which is commonly used in static load balancing, is to estimate the load of each object based on some knowledge of the application, such as the time complexity of the algorithms as a function of input data size and the communication patterns among objects. The advantage of this analytic method is that when the performance model is accurate, it is potentially very responsive to a task with very frequent load changes. In particular, it can predict an upcoming load change and respond to it rather than noticing the change only after it occurs. The disadvantage of this method is that it requires a great effort from the programmer who knows the algorithms best, and it may be quite inaccurate if the performance model is not accurate.

Another way of collecting load data is to measure the load of processors and tasks explicitly. Most machines nowadays provide low overhead clocks with microsecond level accuracy, which can be used to time each task execution. This method also potentially provides an automatic solution for load instrumentation. Since a run-time system such as CHARM++ already schedules object execution and mediates all communication, it can automatically instrument the CPU cost of a task and its communication activities [52]. Thus, measurement-based load balancing has the advantage of being very accurate and requires little programmer effort. The drawback is that measurement-based load balancing strategies rely on an assumption that the past load of a task is a good predictor of its near future. Thus, the accuracy is limited when the load of tasks keeps changing unpredictably.

One can also combine the above two methods to perform measurement-based load collection with the predictive power of the analytic load estimation.

6.4.2 Load Balance Initiation

Due to the overhead of load balancing itself, invoking load balancing with excessive frequency may slow down a program when the cost of load balancing exceeds the possible benefits. Thus, for load balancing to be useful, one must determine *when* to start a new load balancing. Doing so involves two steps: (a) determine if load imbalance occurs, and (b) determine if it is advantageous to start a load balancing by comparing the possible benefits of load balancing and its cost.

Load imbalance can be detected in a synchronous way or in an asynchronous way. In synchronous load imbalance detection, all processors pause their execution at some synchronization points to compute the load imbalance by comparing the individual processor load to the global average load. In asynchronous load imbalance detection, every processor keeps track of a window of load history. There is no synchronization point or barrier to stop for computing the load imbalance. This process occurs in the background while the application continues its execution.

6.4.3 Load Balance Decision Making

Most of the dynamic load balancing strategies can be classified as centralized [72] or fully distributed methods [28, 46, 39]. In centralized strategies, a dedicated processor gathers global information about the state of the entire machine and makes decisions for the migrations of tasks for every processor. In fully distributed strategies, each processor executes load balancing algorithms by exchanging state information with other processors. The migration only happens between neighboring processors.

6.4.4 Object Migration

After load balancing decisions are made, objects are migrated among processors to achieve new load balancing. Object migration must preserve the integrity of object state, including run-time state associated with the object such as buffered messages. The migration of object data typically requires helper functions provided by the application, especially when complex data structures such as link lists or pointers are involved.

6.5 CHARM++ Load Balancing Framework

CHARM++'s object model is particularly well-suited to object-based load balancing. CHARM++ application is written in parallel C++ objects. C++ class promotes data encapsulation, which usually has well-defined regions of memory on which the class operates. This potentially simplifies the packing of data for migration of objects compared to process migration or thread migration, where the thread's entire stack must migrate to a new processor.

Message forwarding after migration is automatically handled by the CHARM++ runtime system. From a user's point of view, CHARM++ objects are location independent; messages are usually delivered to objects instead of processors. Thus, there is no processor-specific state that an application writer needs to worry about for object migration; CHARM++ run-time system takes care of the run-time state associated with the migrating objects.

Using the CHARM++ object model, the run-time system treats application objects uniformly by instrumenting the start and end time of each method invocation on the objects, rather than deriving execution time from some application-specific knowledge. This makes the automatic measurement-based load balancing feasible. Further, the CHARM++ run-time system can automatically record object-to-object and collective communication patterns, so that load balancers can access the communication pattern information for making optimal load balancing decisions.

Unlike MPI, CHARM++ can even separate the idle time from communication overhead. In MPI, when one is waiting at a barrier or a message-receive call, all the time spent gets called communication overhead.

However, this often includes idle time, because another processor has not arrived at the barrier or has not sent the message yet. The CHARM++ run-time system can cleanly separate communication overhead from such idle time. With the rich application load and communication statistics, better load balancing decisions can be made.

The first basic version of CHARM++ measurement-based load balancing framework was implemented by Robert Brunner [48, 22, 24, 23]. It has been redesigned and improved by the author and other group members since then for better functionality and extensibility for broader categories of applications and more complicated load balancing requirements.

The CHARM++ load balancing framework is designed to fulfill two major objectives. First, we want to provide a fully automatic migration for parallel applications written using CHARM++ and AMPI. Specifically, for these applications, making use of the load balancing functionality should involve minimal changes to existing application code. This means that the programming model and interface provided by the CHARM++ and load balancing run-time should closely mirror one another.

Our second objective is to allow as much flexibility as possible in the range of load balancing strategies implementable by the framework. To achieve this, we have isolated the application code from the load balancer's decision-making module. Application's execution statistics including computation load and communication pattern are instrumented in detail in the load balancing database. This application-independent representation of object communication graph permits separation of the decision-making process of load balancing from the application-specific code.

6.6 Framework Overview

One observation leading to our automatic *measurement-based load balancing strategy* in CHARM++ is the heuristic principle that we call the "*principle of persistence*", which is an observation that the computational loads and communication patterns between objects (chares) tend to persist over time, even in dynamic applications. In such cases, the recent past is a good predictor of the near future.

The CHARM++ run-time system (RTS) provides an accurate measurement of application load. With the rich application load and communication statistics, better load balancing decisions can be made.

6.6.1 Components

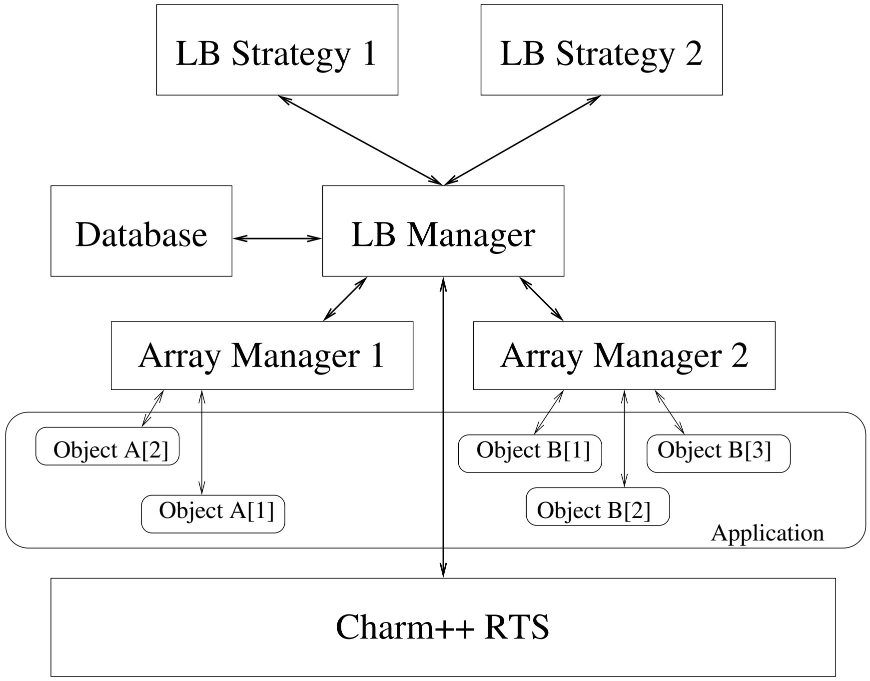


Figure 6.3: Components and interactions in the load balancing framework

Figure 6.3 illustrates the components of the measurement-based load balancing framework on a single processor. At the top level of the figure are the load balancing *Strategies*. Strategies are implemented in CHARM++ as *Chare Groups*. When informed by *LB Manager* to perform load balancing, strategies on each processor may retrieve information from the local LB Manager database about the current state of the processor and the objects currently assigned to it. Depending on a specific strategy, it may communicate with other processors to gather state information. With all information available, strategies determine when and where to migrate object, and provide this information to the LB Manager, which supervises the movements of the objects, informing the strategies as objects arrive. When the moves are complete, the strategies signal the LB Manager to resume the objects. Two types of load balancing strategies are implemented in CHARM++. One is centralized load balancing strategy (in Chapter 7) and the other is fully distributed strategy (in Chapter 8.2).

During execution, the LB Manager monitors the load behavior on each processor. It collects background load and idle time statistics into the LB Database, which is used by the LB Strategies for making load balancing decisions. The LB Manager also interacts with objects through *Array Managers*. Array managers manage object arrays, with one manager created for each array. They are responsible for monitoring the movement of objects and reporting chare objects' arrival and departure to the LB manager. The array manager also monitors the execution of chare objects. When a particular object is being executed, it notifies the LB manager so that the manager may start the timing for its execution. The array manager also reports about communication initiated by the object. The array manager knows how to migrate an object under its control. Thus, when the LB manager receives a request to migrate an object, it forwards the request to the array manager.

The following chapters are organized as follows. In chapter 7, we focus on the centralized load balancing strategies. We evaluate and optimize existing basic centralized strategies for complex needs of applications. The novel contribution of this chapter is the exploration of new load balancing strategies for more complex contexts that arise in real-world applications, such as communication and dependences. We demonstrate with performance case studies that our CHARM++ centralized load balancing strategies work efficiently for a class of applications even for large machines with thousands of processors. In Chapter 8, we further study the limitations of the centralized load balancing strategies for extremely large parallel machines, and present a new effective hierarchical load balancing strategy.

Chapter 7

Centralized Load Balancing Strategies

In centralized load balancing strategies, all instrumented data including objects' computational load and communication pattern on every processor are sent to one particular processor (central processor). Instrumented data are then integrated into an object-communication graph. The number of nodes in the object graph is the number of objects. The weight of each node represents the object's computational load. The edges in the object graph represent the communication between objects and the weight of each edge is denoted as the number of messages sent and the volume of the communication in bytes.

Centralized load balancing applies P-partitioning algorithm (P is the number of processors) to partition the migratable objects into P groups and map the object groups to processors. This load balancing problem is an NP-hard problem. Exhaustive searches are needed to find the optimal solution, which is impractical if the number of objects is large. However, cheaper approximation algorithms with heuristics can often be used to find satisfying solutions.

In CHARM++, a "good" load balancing algorithm involves optimizations for many criteria in the way that migratable objects are mapped to processors. These criteria include communication locality, the critical path and even the cost of a load balancing strategy itself. For real-world applications, load balancing based merely on objects' CPU load and ignoring some of the important criteria such as communication is clearly not enough to achieve near optimal solutions. They can sometimes even lead to degraded performance (an example is in Section 7.4).

In this chapter, we discuss load balancing techniques for doing load balancing based on several important criteria and their performance studies. One important task for a load balancing algorithm is to distribute migratable objects to processors with good *communication locality*. Objects that communicate to each other intensively should be mapped to the same processor or processors that are close to each other. For this purpose, we must consider not only point-to-point communication, but also *collective communication*. Section 7.3 presents such a communication-aware load balancing strategy. *Critical path* is another important constraint for making load balancing decisions. However, it is very difficult or expensive to solve a critical path problem at run-time. In Section 7.4, we propose a load balancing algorithm based on multiple phases of computation to tackle the critical path problem in a specific case. Finally, in Section 7.5, we discuss reducing (or hiding) load balancing overhead by performing asynchronous load balancing to allow overlapping of load balancing time with object computation.

Figure 7.1 illustrates the current centralized load balancers and their class hierarchies that we will present in detail in the following sections. We first present an overview of the basic centralized load balancing strategies, followed by detailed discussion and case studies of more sophisticated load balancing strategies.

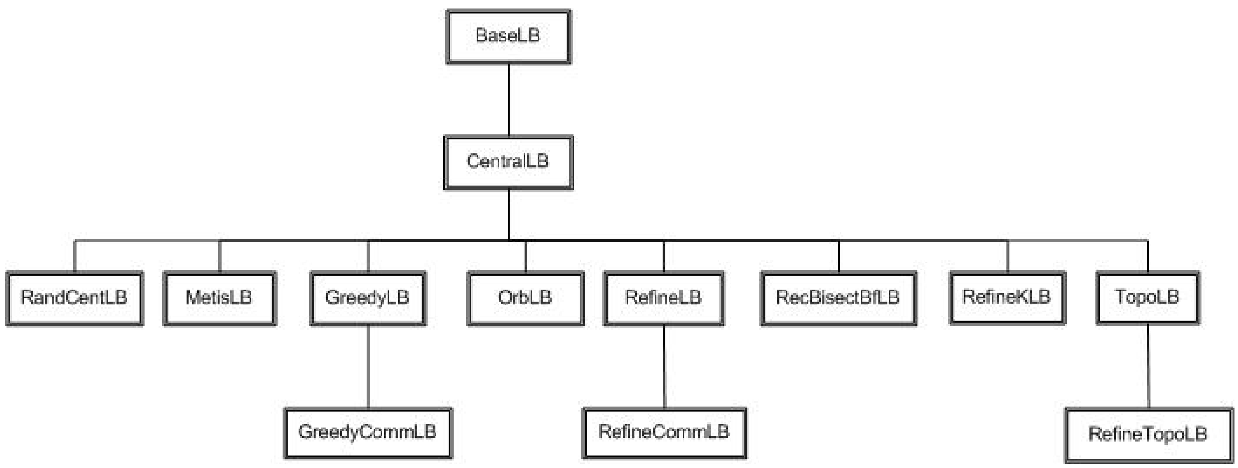


Figure 7.1: Centralized load balancers and their hierarchies

7.1 Quantitative Performance Analysis Methodology

Algorithms with different heuristics may produce different quality of load balance. In general, it is difficult to tell which load balancing strategy is superior to others without running it and comparing the outcome of the load balancing. Thus, it is desirable to quantitatively evaluate the outcome of the load balancing strategies.

Given a parallel system with N processors, each processor i is measured to have compute load of W_i, and the communication cost between node i and node j is denoted as C_{ij}.

For load balancing strategies that do not take communication into account, total CPU load L_i for each processor i is simply:

$$L_i = W_i$$

.

When taking communication into account, the workload on each processor i can be calculated as its computation load W_i with additional communication overhead C_i. The total CPU total becomes:

$$L_i = W_i + \alpha * C_i$$

. α is chosen to reflect how much weight to give for the communication overhead in the total load.

With total CPU load for each processor, some simple but useful load balancing performance metrics can be calculated as following:

- *max load* (o): the heaviest load of all the processors:

$$o = \max_{i=1}^{N} L_i$$

- *average load* (μ): the average load of all the processors:

$$\mu = \frac{1}{N} \sum_{i=1}^{N} L_i$$

- *average imbalance* (σ) : the deviation of the load situation from the ideal balanced one. It is the global standard deviation of the load of all N processors, normalized to the average load (μ):

$$\sigma = \frac{1}{\mu} \sqrt{\frac{\sum_{i=1}^{N} (L_i - \mu)^2}{N}}$$

.

Among these metrics, $maxload$ is the most important one that reflects the quality of a load balancing outcome. This is because a parallel application has to run at the speed of the slowest processor. Thus, a smaller $maxload$ leads to better load balancing result.

To evaluate the performance of a load balancing strategy and compare it against other strategies, one approach is to test these load balancing strategies with synthetic benchmarks or applications and calculate the above performance metrics after load balancing steps. However, the performance result may be easily affected by the dynamic behaviors of the application itself and changing test environment. This makes it difficult to directly compare different load balancing strategies under the same condition. Furthermore, in order to compare load balancing performance of multiple strategies, one has to run the same test programs on a parallel machine many times — each time with a different load balancer, which can be very time consuming if the application needs many processors to run or the execution time is long. For this purpose, we developed a *sequential load balancing simulation tool* to simplify this task.

7.1.1 Sequential Load Balancing Simulation Tool

The sequential load balancing simulation tool allows performance analysis of centralized load balancers *offline* in a sequential fashion (with one processor) without the need to run the parallel application many times with different load balancers.

One simple observation makes this possible, that is when a centralized load balancing starts, the central node gathers all the information from every processor and builds the object-communication graph. Therefore, this object-communication graph can be simply stored to disk in files and retrieved later in a sequential program to load the graph in an offline fashion.

A separate sequential program is written to read the object-communication graph from disk, and invoke the designated load balancing strategy in study with the graph as its input. One can then analyze the outcome of the load balancing strategy easily.

There is some practical concerns in order for the tool to be useful. One concern is that the file stored on one machine should be able to be used on another machine even with different architecture and machine data representations. For example, a database file stored in binary format from a run on LeMieux (a supercomputer comprised of Alphaserver nodes) cannot be directly read on a Linux PC box due to different data representations. To solve this problem, one either has to write the database file in ascii format, or perform data translation across machines if using binary format. We use the CHARM++'s PUP (Pack/UnPack) framework [45] that transparently handles the machine data translation in binary format.

The sequential load balancing simulation tool can even be used to simulate load balancing for a different number of processors, thanks to CHARM++'s processor virtualization. For example, an object-communication graph obtained from a parallel run with only 10 processors can be later used for evaluating a load balancing strategy as if the load balancing runs on a machine with 100 processors. This is due to CHARM++'s virtualization idea - the object-communication graph is processor independent. [1] This mode of load balancing simulation is particularly useful if the access to the large configuration of a parallel machine is difficult.

7.2 Basic Centralized Load Balancing Strategies

Basic centralized load balancing strategies do not take communication into account. They map objects to processors solely based on the object's computation load. There are two categories of algorithms. One is greedy-based algorithms that do not take the existing object mapping into account, and the other is refinement-based algorithms that take the existing mapping into account in order to limit the number of object movements.

7.2.1 Overview

Charm++ implements the following basic centralized load balancers.

RandCentLB

RandCentLB performs random load balancing. It is the simplest load balancers in CHARM++. It does not even use the object graph. It merely picks a random new processor for each object. This can result in reasonable load balance when there are a large number of objects and the communication overhead is small. RandCentLB requires only a single pass through objects, and its computational complexity is $O(N)$, where N is the number of objects.

GreedyLB

GreedyLB is an implementation of a greedy load balancing algorithm. The objects are reassigned regardless of the current processor assignment and communication is not considered. The idea of the greedy algorithm is to pick the heaviest unassigned object and assign it to the currently least loaded processor. Initially, no

[1] In this case, the strategy in study can not rely on the processor-level load data such as background load, idle time.

objects are assigned to any processors, so every processor has no load. When an object is assigned to a processor, the load of the processor is increased by the object load. In the implementation, objects are sorted by their load, and a MinHeap is built for processor load. For N objects, this is an $O(NlogN)$ algorithm.

Algorithm 1: The GreedyLB Algorithm

---

begin

> **Data**: V_t (the set of objects),
>
> V_p (the set of processors),
>
> G_p (the background load of processors)
>
> **Result**: $P : V_t \longrightarrow V_p$ (An object mapping)
>
> // build heap;
>
> $nmigobj \longleftarrow number\_of\_migratable\_objects$;
>
> ObjectHeap objheap(nmigobj+1);
>
> $V_t \longrightarrow maxh\_for\_all\_migratable\_objects$;
>
> MinHeap cpuHeap(P);
>
> //Initially processors are empty with only background load;
>
> $G_p \longrightarrow cpuHeap$;
>
> **for** $i \leftarrow 1$ ***to*** $nmigobj$ **do**
>
> > $o = objheap.deleteMax()$;
> >
> > $donor \longleftarrow cpuHeap- > deleteMin()$;
> >
> > $o \longrightarrow donor$ // assign object o to processor donor;
> >
> > $donor- > load+ = o$ // update donor's load with object load;
> >
> > $cpuHeap- > Heapify()$;

end

RefineLB

RefineLB is an algorithm which improves the load balance by incrementally adjusting the existing object distribution. Refinement is used with an overloaded threshold. Typically, this threshold is set at 1.003 times the average load across all processors. Any processor is considered overloaded if this load is above this threshold. The computational cost of the algorithm is low because only overloaded processors are examined, and it results in only a few objects being migrated.

RefineKLB

RefineKLB [2], as the name suggests, is a refinement strategy to improve load balance by moving at most k objects from their processors. The motivation of this strategy is that a reassignment from scratch relocates almost all objects leading to a large overhead of moving object-data across processors. In contrast, RefineKLB keeps most of the objects at their original location while moving only a bounded number (k) to achieve load balance. The parameter k can be passed as a percentage of total objects at runtime. The strategy implements the load rebalancing algorithm in [7] with minor adaptation to account for background load on processors. This gives a 1.5-approximation guarantee with respect to the best possible load balance achievable by moving up to k objects. The metric to measure load balance is the total load on the most-loaded processor. The running time of the algorithm is $O(NlogN)$ where N is the number of objects, independent of the value K.

OrbLB

OrbLB treats objects with spatial coordinates. It applies an orthogonal recursive bisection algorithm which attempts to provide a more balanced division of space. For example, in two dimensions the first division finds a vertical line that divides the objects into two groups with approximately equal load. The second division finds a horizontal line that divides each of the two groups into two equally loaded groups. The subdivision continues until either the number of groups equals the number of processors or there is only one object in each group.

7.2.2 Performance Evaluation

We ran tests to compare the outcome of various basic centralized load balancing algorithms in some of the performance metrics described in Section 7.1.

We ran Jacobi2D on both 64 and 1024 processors of LeMieux and dumped the object-communication graph using the sequential load balancing simulation tool. For each set of data, we compared the perfor-

[2] RefineKLB is implemented by Tarun Agarwal.

mance metrics (Section 7.1) for various load balancing strategies. The result is shown in Table 7.1. For each row of a load balancing algorithm, it shows the minimum and maximum processor load, average load after load balancing and the time taken for the algorithm, for both data sets. The first row shows the metrics data before load balancing.

	64 processors (2048 objs, 0.372MB)				1024 processors(10240 objs, 1.904MB)			
	Min	Max	Ave Load	Time	Min	Max	Ave Load	Time
—	13.952	15.505	14.388	-	42.801	45.971	44.784	-
GreedyLB	14.063	14.425	14.344	0.0978	43.199	44.829	44.776	0.1389
GreedyRefLB	14.063	14.425	14.344	0.1436	43.199	44.829	44.776	0.244
RefineLB	14.141	14.647	14.387	0.095s	42.801	45.971	44.784	0.25s
OrbLB	11.350	12.414	11.891	0.051s	31.269	44.940	38.20	0.37s

Table 7.1: Performance of Basic Centralized Load Balancers (Jacobi)

As the result shows, OrbLB is doing particularly well for this Jacobi program due to OrbLB's geometic partitioning algorithm that happens to optimize for the communication locality the best in this application.

We also run a load balancing benchmark program (lb_test) to evaluate the basic load balancing strategies. In the following tests, lb_test program creates 10240 parallel objects on 1024 processors. These objects communicate in a mesh 2D or a random graph virtual topology. Random graph is constructed with random communicating edges between objects and roughly 25% of links is used. Table 7.2 and 7.3 show the load balancing quality of various load balancing strategies for mesh2D and random graph virtual topology respectively. OrbLB does very well for the mesh2D virtual topology, however performs poorly for the test with random graph virtual topology due to the mismatch of the geometric information and actual communication pattern.

	max load	comm volume	comm load	Time
—	0.00731	77863K	0.007326	-
GreedyLB	0.00638	80100K	0.007686	0.0272
RefineLB	0.00671	77908K	0.008059	0.0698
OrbLB	0.007472	45122K	0.005122	0.342

Table 7.2: Performance of Centralized Load Balancers (mesh2d communication)

We run a real-world finite element application (simulating crack propagation) with various basic load balancing strategies. These tests involve 1000 AMPI threads running on 100 processors. The mesh in simulation comprised of 91292 cohesive elements at the interface plane and 4198134 linear strain tetrahedral elements. The result is shown in Table 7.4 for comparison of various basic load balancing strategies. OrbLB

	max load	comm volume	comm load	Time
—	0.168190	2011M	0.206238	-
GreedyLB	0.145382	2013M	0.201103	0.0268
RefineLB	0.143584	2011M	0.207520	0.0346
OrbLB	0.1597	1996M	0.227844	0.325

Table 7.3: Performance of Centralized Load Balancers (randgraph communication)

in this case does fairly well in optimizing communication, however, it does not balance the load as well as the other strategies in reducing the maximum load.

	max load	comm volume	comm load	Time
—	105.092349	7755M	3.444740	-
GreedyLB	94.282370	14254M	3.696606	0.00205
RefineLB	99.424751	7973M	3.444740	0.00134
OrbLB	107.179463	7795M	3.030202	0.0181

Table 7.4: Performance of Centralized Load Balancers (Fractography FEM)

7.2.3 Application Study — NAMD

Basic centralized strategies are found to be useful for many applications when communication is not a significant concern. In this section, we present case studies of load balancing for several real-world applications using these basic strategies.

NAMD is a parallel production quality molecular dynamics program written in Charm++ and designed for high performance simulation of large biomolecular systems on large parallel machines.

The tests in this section were conducted on PSC LeMieux, a 750 node, 3000 processor cluster. Each node in LeMieux is a Quad 1Ghz Alpha server connected by Quadrics Elan, a high speed interconnect with 4.5μs latency.

Dynamic load balancing was an important performance challenge for this application. The distribution of atoms over space is relatively non-uniform, and the computational work is distributed quite non-uniformly among the objects. We used the Charm++ measurement-based load balancing framework, which supports runtime load and communication tracing. The run-time system admits different strategies (even during a single run) as plug-ins, which use the trace data. We used a specific greedy strategy [51]. For a 128-processor run, Projections visualization of the utilization graph (Fig. 7.2(a)) confirmed that the load balancer worked very well: Prior to load balancing (at 82 seconds) load imbalance led to utilization averaging to 65-70% in

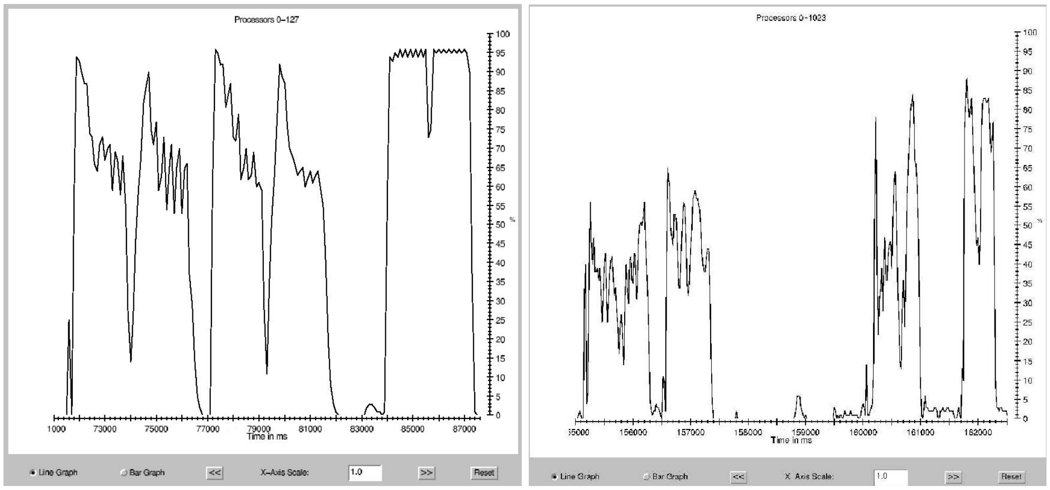


Figure 7.2: Average processor utilization against time on (a) 128 (b) 1024 processors

each cycle. However, after load balancing, the next 16 steps ran at over 95% utilization.

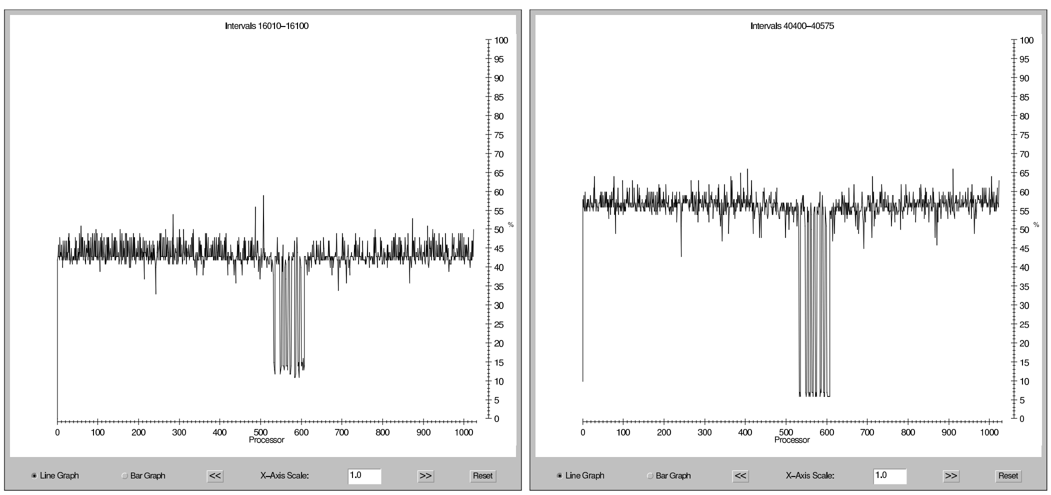


Figure 7.3: Average Processor Utilization after (a) greedy load balancing and (b) refining

However, when the same strategy was used on 1024 processors, the results were not as satisfying (Figure 7.2 (b)). In particular, (via a profile view not shown here) it became clear that the load on many processors was substantially different than what the load balancer had predicted. Since the greedy strategy used ignored existing placements of objects entirely (in order to create an unconstrained close-to-optimal mapping), it was surmised that the assumptions about background load (due to communication, for example) as well as cache performance were substantially different in the new context after the massive object migration induced by load balancer. Since the new mapping was expected to be close to optimal, we did not want to discard it. Instead, we added another load balancing phase immediately after the greedy reallocation, which used a simpler "refinement" strategy: objects were moved only from the processors that were well above (say 5%) the average load. This ensured that the overall performance context (and communication behavior) was not perturbed significantly after refinement, and so the load-balancer predictions were in line with what happened. In Figure 7.2 (b), the initial greedy balancer works from 157 through 160 seconds, leading to some increase in average utilization. Further, after the refinement strategy finished (within about .7 seconds) at around 161.6 seconds, we can see that utilization is significantly improved. Another view in Projections (Fig. 7.3), showing utilization as a function of processors for the time intervals before and after refinement, shows this effect clearly.

Note that due to some quirks in the background load, several processors in the range between 500 and 600 were left underloaded by the greedy algorithm. The refinement algorithm did not change the load on those, since it focuses (correctly) only on overloaded processors: having a few underloaded processors does not impact the performance much, but having even one overloaded processor slows the overall execution time. Here, we see that 4 processors (e.g, processor 508) were significantly overloaded before the refinement step, whereas the load is much closer to the average after refinement. As a result, overall utilization across all processors rises from 45% to 60%.

Stretches in Entry Methods

This problem was first found while running NAMD on a large number of processors on LeMieux, particularly when using all four processors of a node. Figure 7.4 shows the timeline of NAMD on 1536 processors. Observe that processors 900 (processor 6 from the top) and 933 (processor 7) have handlers that last about 20-30 ms. This is clearly shown by the long superscript bar (colored in light grey) on top of the handler, which shows a send operation. Both the stretched handlers here block on a send operation. Normally these handlers should take about 2-3ms to finish, as shown by the remaining rectangles. We believe these stretches were caused by a mis-tuned Elan library and operating system daemon interference.

We reported the problem to Quadrics and they provided a fix in the Elan library that alleviated the problem. Fixing the Elan software, however, did not completely eliminate stretches. The interference of operating system is substantial, considering the fact that NAMD simulation of the ATPase system takes only about 12ms per step on 3,000 processors. The step time is very close to the 10ms time quanta of the operating system. This, if a heavy daemon (such as a file system daemon) is scheduled on any of the 3,000

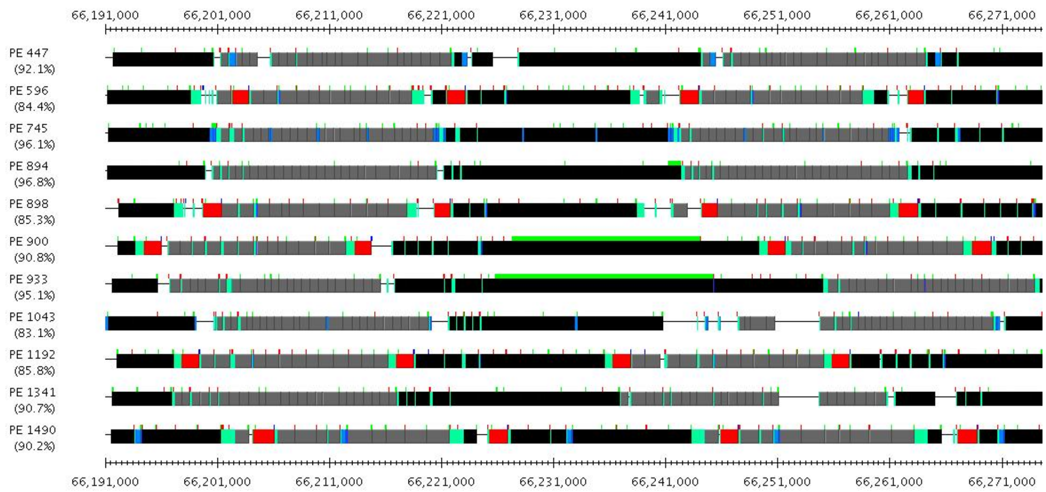


Figure 7.4: NAMD Timineline view on 1536 processors

processors, NAMD step time could be affected noticeably. This operating system daemon interference on LeMieux is not unique, we observed the similar stretches on Turing clusters of CSE. Turning off some of the daemons on compute nodes did help improving the performance.

Petrini et al. have studied this issue of operating system interference [29] in great detail. They present substantial performance gains for the SAGE application on ASCI-Q (a QsNet-Alpha system similar to LeMieux) after certain file system daemons were shutdown.

Since we do not have control over the machine to conduct the experiments with daemons, we tried several strategies to tolerate the interference at application level. One idea is to use blocking receives instead of busy waiting, which gives the operating system time for scheduling daemons. Another idea we experimented was **sub-step load balancing**, which is a combination of measurement-based load balancing for iterative applications and task scheduling load balancing. A measurement-based load balancer predicts the future load for the computation in the next few iterations and broadcasts the predicted load to every processor. A load balancing framework on each processor constantly monitors its work load while computation moves forward. When the measured load is greater than the predicted load to a certain extent, it indicates that a stretching happens.

In response to the stretching, an application running on the stretched processor starts to send some of its computation work to other idle or underloaded processors. It does so by generating a special type of one-time lived small tasks in a message sent to a remote processor. This load balancing framework then

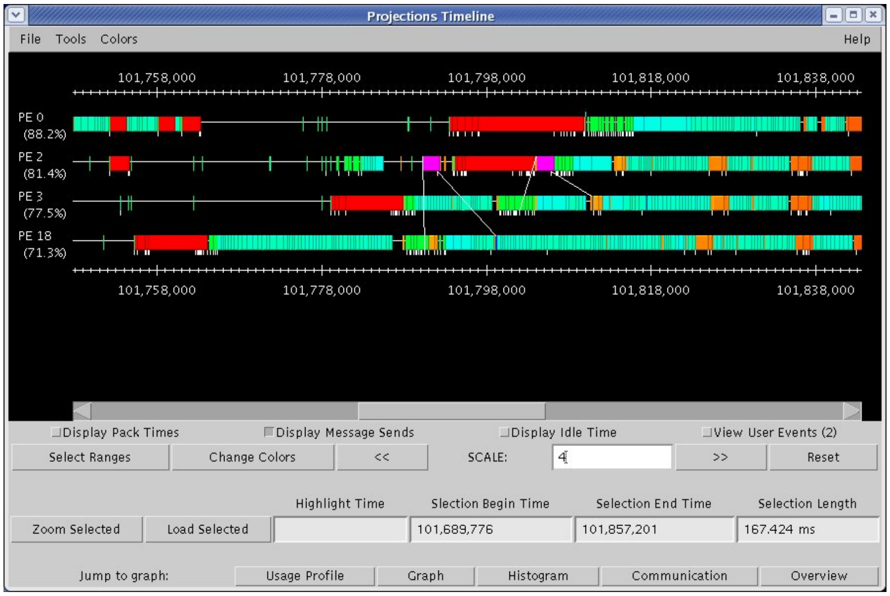


Figure 7.5: NAMD sub-step load balancing with ApoA1 benchmark on Turing

takes these tasks and schedules them to some underloaded processors using the same techniques that are used in the *seed load balancer*.

We did some tests of NAMD with sub-step load balancing on both LeMieux and Turing cluster. A Projections timeline view in Figure 7.5 illustrates how sub-step load balancing is performed. When stretching occurs on processors 3 and 13, both processors send a task to processor 2. When these one-time lived tasks finish their computation on processor 2, they send the computation result back to its original processors and remove themselves.

NAMD run with sub-step load balancing on 2250 processors of LeMieux did not obtain noticeable improvement in its performance. With ATPase benchmark, NAMD time per step is 14.67ms in a normal run as opposed to 14.53ms with sub-step load balancing. This was partly due to the shutdown of daemons on the machine, so that many stretches no longer happened. However, the result shows that the sub-step load balancing was efficient (at least did not degrade the performance) despite of the overhead of sending transient tasks to remote processors.

It can be anticipated that when parallel machines grow in size, more and more applications will run fast enough to be vulnerable to any kind of interference from operating system daemons. Research in both operating systems and application techniques is needed to overcome this stretching problem in order for applications to scale to very large machines with sustainable performance.

Summary

NAMD won the Gordon Bell award at SC2002 with unprecedented speedup on 3,000 processors on Pittsburgh Supercomputing Center's LeMieux supercomputer [3] with teraflops level peak performance.

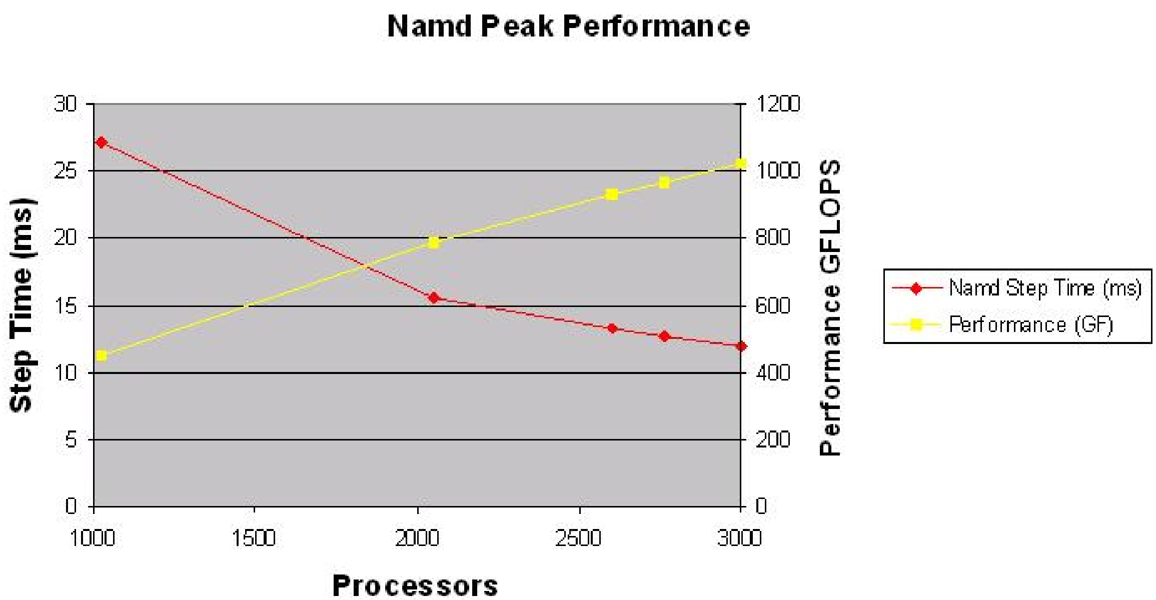


Figure 7.6: NAMD Performance on 327K atom ATPase PME benchmark, on PSC LeMieux

Figure 7.6 shows the scalability of the ATPase NAMD simulation to 3000 processors on the PSC LeMieux machine. Both step time and floating point performance are shown in the plot. The best achieved step time is 12ms with a floating point performance over one TF.

NAMD scales not only on several thousands of processors of LeMieux, but also on various other platforms from supercomputers to commodity clusters.

Figure 7.7 [4] illustrates the portable scalability of NAMD on a variety of platforms with load balancing. Each curve represents total resources (processors multiples by time per step) consumed per step for the ApoA1 PME benchmark by NAMD on varying numbers of processors for a specific parallel platform. Perfect linear scaling is a horizontal line. Diagonal scale shows runtime per ns, representing absolute performance — the time to solution as experienced by the user. As the figure demonstrates, CHARM++'s automatic load balancing framework achieves high performance on various platforms without the need to

[3] 750 Quad 1Ghz Compaq Alphaserver ES45 nodes connected by a Quadrics highspeed network

[4] The figure is also available at NAMD official website: http://www.ks.uiuc.edu/Research/namd/performance.html

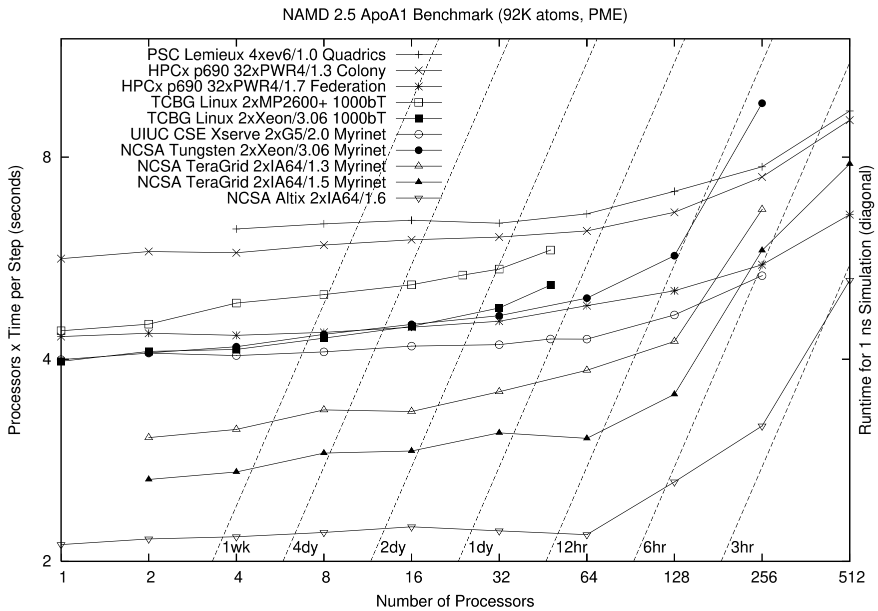


Figure 7.7: NAMD Performance on various platforms with ApoA1 benchmark

specifically optimize for a certain platform; the load balancing run-time automatically adapts to the execution environment to achieve optimal performance.

7.3 Load Balancing with Communication

These load balancing strategies are more sophisticated strategies that take communication into account, including both the point-to-point and collective communication. The basic idea in these strategies is balancing the load by partitioning the object-communication graph into P partitions where P is the number of processors, so that most communicating objects are likely to be placed on one processor.

7.3.1 Overview

The following strategies in Charm++ belong to this category.

GreedyCommLB

GreedyCommLB is an algorithm similar to GreedyLB but taking communication into account. When making assignment for a heaviest object, it tests against not only the least loaded processor, but also the processors the object communicates with, to find the best possible choice. The algorithm is described in 2.

Algorithm 2: The GreedyCommLB Algorithm

begin

Data: V_t (the set of Tasks),

E_o (communication graph)

V_p (the set of processors),

G_p (the background load of processors)

Result: $P : V_t \longrightarrow V_p$ (A task mapping)

// build heap;

$nmigobj \longleftarrow number\_of\_migratable\_objects$;

ObjectHeap objheap(nmigobj+1);

$V_t \longrightarrow maxh\_for\_all\_migratable\_objects$;

ProcessorHeap lightProcessors(P);

//Initially processors are empty with only background load;

$G_p \longrightarrow lightProcessors$;

for $i \leftarrow 1$ ***to*** $nmigobj$ **do**

$minLoad = MAX\_DOUBLE$;

$o = objheap.deleteMax()$;

//find donor;

$donor \longleftarrow lightProcessors- > deleteMin()$;

$comm\_cost \longleftarrow \sum_{e_{ob} \in E_o \wedge P(b)=donor} \hat{c_{ob}}$;

$new\_load = G\_donor.load + comm\_cost$;

if $new_load < minLoad$ **then**

$minLoad = new_load$;

$newP = donor$;

for $donor \in P\_comm$*(P_comm is a set of processors object o communicates)* **do**

$comm\_cost \longleftarrow \sum_{e_{ob} \in E_o \wedge P(b)=donor} \hat{c_{ob}}$;

$new\_load = G\_donor.load + comm\_cost$;

if $new\_load < minLoad$ **then**

$minLoad = new\_load$;

$newP = donor$;

$o \longrightarrow donor$ // assign object o to processor donor;

end

Assuming each object has a constant number of processor it communicates with, for N objects and P processors, the complexity of the algorithm is:

$$O(N(logN + logP) + PlogP)$$

If $N >> P$, the complexity is dominated by N, thus the complexity of the algorithm is $O(NlogN)$.

RefineCommLB

RefineCommLB is an algorithm similar to RefineLB but taking both point-to-point and collective communication into account.

MetisLB

MetisLB uses the METIS graph partitioning library [56] to partition the object-communication graph with node (object) weights and communication loads on edges. METIS uses a multilevel k-way hypergraph partitioning algorithm to partition the graph. METIS is intended for finite element mesh generation with graphs of hundreds of thousands to millions of nodes and with graph degrees of approximately 10. Thus, for small load balancing problems, it may not work as well as simpler schemes.

RecBisectBfLB

RecBisectBfLB uses the object communication graph to recursively partition the objects until there is one partition for each processor. Initially, the graph is partitioned into two pieces by growing regions around two arbitrary nodes in the graph. Two interleaved breadth-first searches are carried out from the two initial nodes, with the region having the lowest assigned load being expanded by one node at each iteration. Once two partitions are formed, each partition may be further partitioned recursively until the number of partitions obtained is equal to the number of processors.

7.3.2 Performance Evaluation

We repeat the tests in Section 7.2.2 to compare the outcome of various centralized load balancing algorithms in some of the performance metrics described in Section 7.1.

We ran the same Jacobi2D using the same load balancing data on both 64 and 1024 processors to evaluate the new communication-aware load balancing strategies. The result is shown in Table 7.5. For each row of a load balancing algorithm, it shows the minimum and maximum processor load, average load after load balancing and the time taken for the algorithm, for both data sets. The first row shows the metrics data before load balancing.

	64 processors (2048 objs, 0.372MB)				1024 processors(10240 objs, 1.904MB)			
	Min	Max	Ave Load	Time	Min	Max	Ave Load	Time
—	13.952	15.505	14.388	-	42.801	45.971	44.784	-
GreedyLB	14.063	14.425	14.344	0.0978	43.199	44.829	44.776	0.1389
GreedyRefLB	14.063	14.425	14.344	0.1436	43.199	44.829	44.776	0.244
GreedyCommLB	13.748	14.396	14.025	0.0913	40.519	46.922	43.777	4.36
RecBisectBfLB	11.701	13.771	12.709	0.179	35.907	48.889	43.953	4.409
MetisLB	14.061	14.506	14.341	0.32	41.477	48.077	44.772	73.43
RefineLB	14.141	14.647	14.387	0.095s	42.801	45.971	44.784	0.25s
RefineCommLB	14.143	14.629	14.388	0.118s	42.801	45.971	44.784	0.386s
OrbLB	11.350	12.414	11.891	0.051s	31.269	44.940	38.20	0.37s

Table 7.5: Performance of Communication-aware Centralized Load Balancers (Jacobi)

As the result shows, those strategies that take communication into account generally are much slower. For example, it takes about 73 seconds for MetisLB to balance the load of this application on 1024 processors. It can also be seen that OrbLB is doing particularly well for this Jacobi program due to OrbLB's geometic partitioning algorithm that happen to optimize for the communication locality the best in this application. MetisLB, although uses a much more complicated algorithm optimizing communication explicitly, does not deliver as good load balancing results.

We also repeat the tests with benchmark program lb_test but with comparison to new communication-aware load balancing strategies. Table 7.6 and 7.7 show the load balancing quality of various load balancing strategies for mesh2D and random graph virtual topology respectively. For tests with simple mesh2D virtual topology, MetisLB appears to work very well and optimizes communication dramatically. However, for complex communication pattern as in random graph, MetisLB does not perform as well as other communication-aware load balancing strategies — it reduces communication volume better, but fails to reduce the communication overhead [5].

Finally we repeat the test with the finite element application with comparisons of various communication-aware load balancing strategies. The result is shown in Table 7.8. Similar to OrbLB, MetisLB in this case

[5]Communication overhead is calculated not only based on communication volume, but also based on number of sends

	max load	comm volume	comm load	Time
—	0.00731	77863K	0.007326	-
GreedyLB	0.00638	80100K	0.007686	0.0272
GreedyCommLB	0.00825	70162K	0.00732	0.088
RecBisectBfLB	0.006876	46920K	0.006587	0.360
OrbLB	0.007472	45122K	0.005122	0.342
MetisLB	0.0080	28661K	0.004023	1.762

Table 7.6: Performance of Communication-aware Centralized Load Balancers (mesh2d communication)

	max load	comm volume	comm load	Time
—	0.168190	2011M	0.206238	-
GreedyLB	0.145382	2013M	0.201103	0.0268
GreedyCommLB	0.168523	2006M	0.197073	3.64
RecBisectBfLB	0.14837	2002M	0.205132	3.151
OrbLB	0.1597	1996M	0.227844	0.325
MetisLB	0.1707	1985M	0.228943	5.756

Table 7.7: Performance of Communication-aware Centralized Load Balancers (randgraph communication)

does fairly well in optimizing communication, however, it does not balance the load as well as the other strategies in reducing the maximum load.

	max load	comm volume	comm load	Time
—	105.092349	7755M	3.444740	-
GreedyLB	94.282370	14254M	3.696606	0.00205
GreedyCommLB	94.980593	13906M	3.646116	0.0142
RefineLB	99.424751	7973M	3.444740	0.00134
RecBisectBfLB	100.829725	11554M	3.506918	0.0134
OrbLB	107.179463	7795M	3.030202	0.0181
MetisLB	117.722816	7157M	3.100350	0.0654

Table 7.8: Performance of Communication-aware Centralized Load Balancers (Fractography FEM)

7.3.3 Collective Communication

In Charm++, object collective communication is message passing that involves a collection of migratable objects in a collective communication function. These functions provide capabilities such as:

- Broadcasting — the transmission of messages from one object to all other objects in a specific chare array;
- Multicast — one-to-many collective communication where one object sends a message to a subset of

objects in an object array;

- Reduction operations (e.g., the summation of data elements contributed from a group of objects)

Note that unlike MPI collective communication that involves a collection of processors, Charm++ collective communication is based on migratable objects. Another difference from MPI is that all collective communication in Charm++ is asynchronous, which permits overlapping of communication with computation.

Broadcast in general is not a very important factor to load balancing algorithms especially when there is at least one migratable object on each processor, since all objects (hence all processors) are involved in the communication. Multicast, however, is useful to load balancing in that communication only occurs in a subset of objects. Charm++ implements a *section multicast/reduction* library to optimize such type of communication.

Section Multicast/Reduction

An *array section* is defined as a subset of with an array of migratable objects of same type. An *array section multicast* is the one-to-many delivery of the same message from one object (the multicast root) to the members in the array section. *Section reduction* is the reduce operation (such as sum, max, logical AND, etc.) performed across all the members of a section. It is like global array reduction, but only occur on a subset of an object array.

Charm++ section multicast/reduction library is implemented as a tree-based communication structure as illustrated in Figure 7.8. Using a binary tree as an example, a message sent from the multicast root (the black circle object) traverses along a pre-built binary tree that spans all the processors that have at least one array section member. On each processor, a multicast manager acts like a proxy to handle the relaying of messages along the tree and the delivery of messages to all local array section members. Compared with direct one-to-many multicast implementation, using the spanning tree optimizes the multicast/reduction communication by sending only one message to a processor that has a section member. Messaging overhead is also distributed among processors in the tree instead of imposing bottlenecks on the multicast root processor.

Section multicast and reduction is widely used in many real-world applications. For example in the LeanMD (Section 5.1) molecular dynamic simulation program, the communication among Cell and CellPair is implemented in section multicast and reduction. Cell objects as section multicast roots send atom position

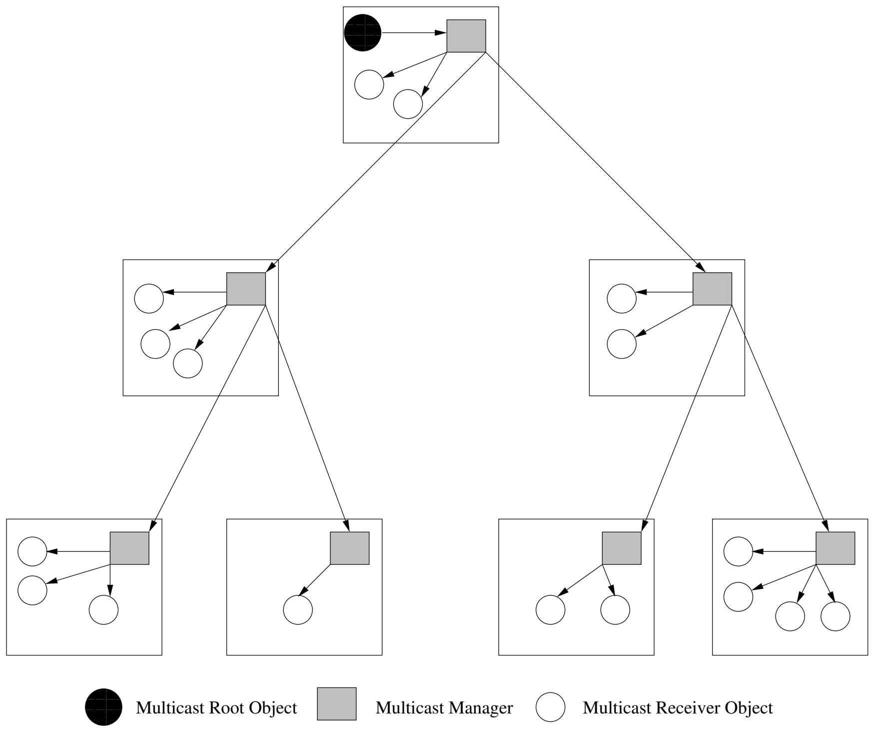


Figure 7.8: Section multicast via a binary tree

data to a set of CellPair objects that they interact with via a section multicast. CellPair objects send back force messages to Cell objects; these force messages are summed up via section reduction (accumulation is one of the combine operations supported) and the final force message is delivered to the Cell object.

The section multicast/reduction library is modified to interact with the load balancing instrumentation module to record multicast and reduction operations as special collective calls. This is necessary because multicast and reduction operations are implemented using a spanning tree. Without the knowledge about the section multicast and reduction operation, the original instrumentation module simply treated the collective operations as individual point-to-point messages sent along the tree. Therefore, the load balancing database not only loses the information about collective operations, but also tends to make incorrect load balancing decisions based on the point-to-point communication on the tree.

With multicast and reduction operations recorded as special collective calls in the object-communication graph, load balancing strategies can then make well informed object mapping decisions. One great property of section multicast/reduction to load balancing is that when there is already a section member residing on a processor, placing another section member on the processor does not incur much additional communication overhead for that processor. Further, it helps to reduce communication overhead by reducing the size of the spanning tree. Based on this heuristics, the load balancing algorithm can optimize the object-to-processor mapping for the collective communication to achieve better performance.

A case study of LeanMD using section multicast library and its load balancing is presented next.

7.3.4 Application Study — LeanMD

The benchmark we used is the Human Carbonic Anhydrase (HCA) dataset. This HCA system is a medium sized atom system which has 30652 atoms, 2135 bonds, 3833 angles, 11769 torsions and 10798 1-4 interactions. The tests were conducted on LeMieux at PSC.

All the simulation results are for the cut-off simulation with a cutoff of 10 Å. Atoms migrate between cells every 5 timesteps.

LeanMD has an option to use *array section multicast library* (Section 7.3.3) to optimize the multicast and reduction communication among a subset of array elements.

Initial runs of LeanMD on LeMieux showed bad scalability beyond 32 processors due to load imbalance problems. Table 7.9 (without section multicast library) and 7.10 (with section multicast library) shows the result of 1-away LeanMD execution before load balancing. It shows that the runs with section multicast library perform better than those without it. However, in both cases, the application could not scale to even a hundred processors.

Processors	Average time per step(sec)	Speedup
1	10.397201	1.00
2	5.409972	1.92
4	2.919763	3.56
8	1.584468	6.56
16	0.962452	10.80
32	0.557294	18.66
64	0.400316	25.97
128	0.200286	51.91
256	0.102508	101.43
512	0.079838	130.23

Table 7.9: Execution times for 1-away LeanMD simulation before load balancing. These simulations do not use the section multicast library. Time per step is taken as the average over five simulation steps.

We experimented with refinement-based load balancers, but the result was not as good as with the load balancers using greedy algorithms. This was because refinement-based load balancers tend to rely

Processors	Average time per step(sec)	Speedup
1	10.345	1.00
2	5.200	1.99
4	2.699	3.83
8	1.421	7.28
16	0.810	12.77
32	0.428	24.17
64	0.321	32.23
128	0.165	62.70
256	0.089	116.24
512	0.076	136.12

Table 7.10: Execution times for 1-away LeanMD simulation before load balancing. These simulations use section multicast library. Time per step reported in this table is average over five simulation steps.

Processors	Average time per step(sec)	Speedup
1	10.397201	1.00
2	5.511004	1.89
4	2.816147	3.69
8	1.425	7.30
16	0.723	14.38
32	0.368	28.25
64	0.199	54.24
128	0.101	102.94
256	0.052	199.95
512	0.036	288.81

Table 7.11: Execution times for 1-away LeanMD simulation with load balancing. These simulations do not use the section multicast library.

on the existing load distribution. They perform well when load is almost balanced, but they may show little improvement when there is severe load imbalance. In LeanMD, objects initially are only mapped to processors in a round-robin fashion, without careful balance of the load. Thus, a more aggressive load balancer is desireable to re-map the objects to processors.

Results without section multicast optimization after load balancing (with Greedy strategy) are summarized in table 7.11. Figure 7.9 shows the Projections overview graph for a 64 processor run. This figure shows that after load balancing, computation load was almost equally distributed between processors.

Table 7.12 shows the result with section multicast optimization and the same GreedyLB as in Table 7.11. In both cases, GreedyLB does not consider communication. Table 7.13 shows the result with section mul-

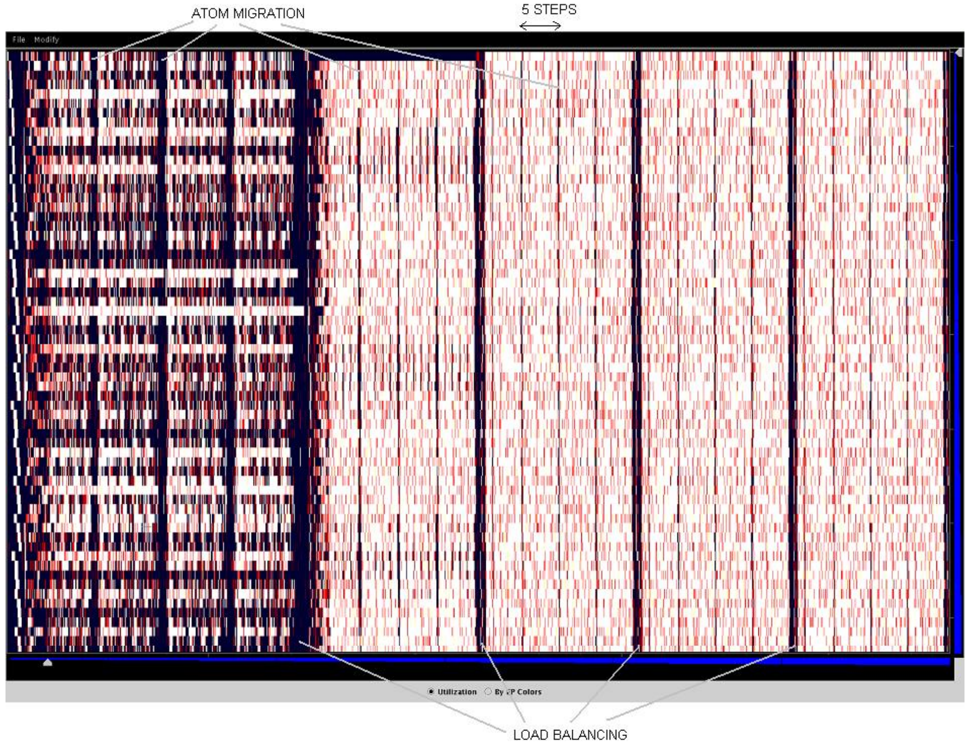


Figure 7.9: Overview of a LeanMD 1-away simulation on 64 processors with GreedyLB.

ticast optimization and with GreedyCommLB which optimizes the communication including the multicast. Compared with the results in Table 7.12, it can be seen that GreedyCommLB with the optimization in improving communication locality outperforms GreedyLB.

7.4 Temporal Characteristic - Multi-phase Computation

Original load balancing strategies make the following assumption about the computation pattern of the objects being balanced: objects remain active during the *entire* execution, and the computation is free of *critical tasks*. However, this may not be realistic in many applications. For example, in LeanMD, the force integration step of Cell objects must be performed before each Cell can send their atom data to the CellPairs. This means that there are two separate phases during the execution, one being force integration and the other being force calculation by CellPairs. In fact, the temporal relationship among objects' computation phases creates critical paths which must be executed "sequentially" in certain orders.

In a different scenario, objects may be doing computation in one period of time, but become dormant during another period of time. For example, the PME calculation step in LeanMD may only occur once in

Processors	Average time per step(sec)	Speedup
1	10.345	1.00
2	5.250	1.97
4	2.710	3.82
8	1.422	7.27
16	0.732	14.13
32	0.375	27.59
64	0.192	53.88
128	0.103	100.44
256	0.056	184.73
512	0.035	295.57

Table 7.12: Execution times for 1-away LeanMD simulation after load balancing (Greedy) using section multicast library. Time per step reported in this table is average over fifteen simulation steps.

Processors	Average time per step(sec)	Speedup
1	10.345	1.00
2	5.248	1.97
4	2.720	3.80
8	1.423	7.27
16	0.730	14.17
32	0.368	28.11
64	0.190	54.45
128	0.099	104.49
256	0.053	195.19
512	0.033	313.48
1024	0.026	397.88

Table 7.13: Execution times for 1-away LeanMD simulation after load balancing (GreedyCommLB) using section multicast library. Time per step reported in this table is average over fifteen simulation steps.

every four steps. Therefore, in such application, there are two different phases — one phase with additional PME objects working on the PME calculation, the other phase without. Trying to balance the load from all objects together may lead to very poor load balancing decisions. Instead, the PME objects need to be balanced separately in order to ensure that the load is balanced in non-PME objects in the three non-PME steps as well.

In both scenarios, recognizing temporal characteristics of objects is essential for load balancers to make correct load balancing decisions. Consider the following scenario as illustrated in Figure 7.10: two different types of objects A and B are doing computation during a period of time. It happens that B objects depend on the work of A objects, thus B objects can only start working after A objects finish. Without knowing

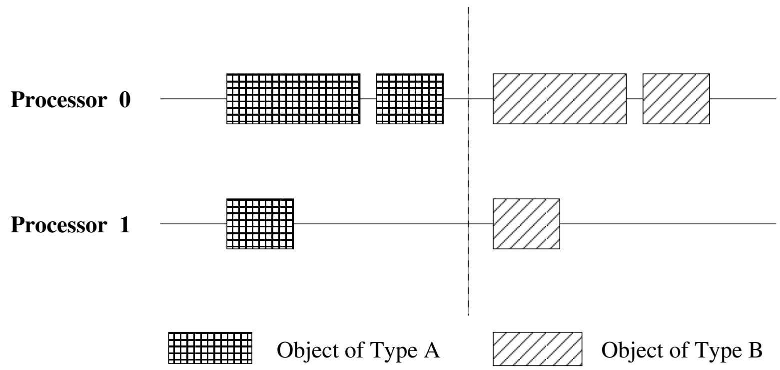


Figure 7.10: Original timeline with mapping of objects on 2 processors

the temporal characteristics of these two objects, a load balancing algorithm could make very poor load balancing decision like in Figure 7.11. In this case, processor 0 has all the A objects and processor 1 has all the B objects. Although here the total loads of both processors are equal and hence balanced in the view of the entire execution, the load actually is not balanced in both critical phases, which leads to degraded application performance.

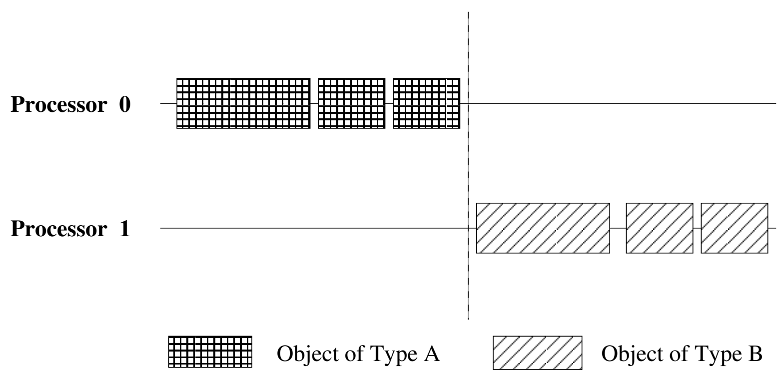


Figure 7.11: Timeline with bad mapping of objects on 2 processors after load balancing

Taking critical path into account is a very difficult load balancing problem that requires a lot of effort. For example, task dependence graphs and Gantt charts [] need to be built for load balancers in order to keep track of the order in which computation is performed. After load balancing occurs, the execution order needs to be enforced by the run-time system to fulfill the load balancing decisions.

We tackle this problem by partitioning the critical path into phases as in the previous examples. To avoid the run-time complexity involved in detecting the critial paths or phases, we provide users an API to specify the boundary of phases. In such a *multi-phase load balancing* algorithm, each phase is balanced separately. For the same example in Figure 7.10, a multi-phase load balancing could produce a better object-

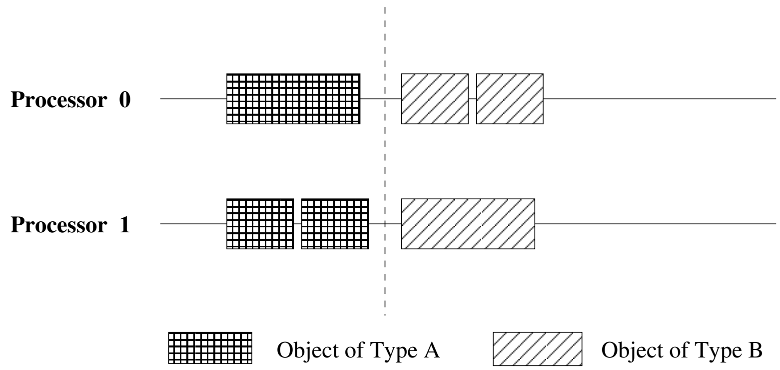


Figure 7.12: Timeline with better mapping of objects on 2 processors after load balancing

to-processor mapping as shown in Figure 7.12. We implemented this idea in PhaseLB on the Charm++ load balancing framework.

7.4.1 Experiments

To evaluate the performance of PhaseLB, we implemented a test program to mimic the typical parallel application that involves computation phases, with flexible control of the amount of computation in each phase. The *phased-iter* program is described below, followed by performance evaluations of PhaseLB and the comparisons with other load balancers that do not consider phase.

The *phased-iter* program performs a iterative computation. In each iteration, there are two computation phases involved, and there are three types of object arrays created to perform phases computation. object array A works only in phase 1, and object array B works only in phase 2, while object array C works continuously in both phases.

The program was run on 32 processors of Tungsten Myrinet cluster at NCSA. To demonstrate the benefit of the phase load balancing strategy, we compared the result of running phased-iter (a) without load balancing, (b) with greedy load balancer (GreedyLB) and (c) with phase-by-phase load balancer (PhaseLB).

Figure 7.13 illustrates the execution time in seconds per phase over a total of 64 steps of all the three cases. It can be seen that in the run without load balancing, in each time step the first phase takes longer than the second phase. The detailed Projections views of utilization over time and across processors are shown in Figure 7.14, where in Figure 7.14(b) load imbalance can be seen across processors. When applying GreedyLB, the performance was actually degraded, especially in the time of the second phase as the small dotted line shows in Figure 7.13. The detailed Projections views of utilization over time and across

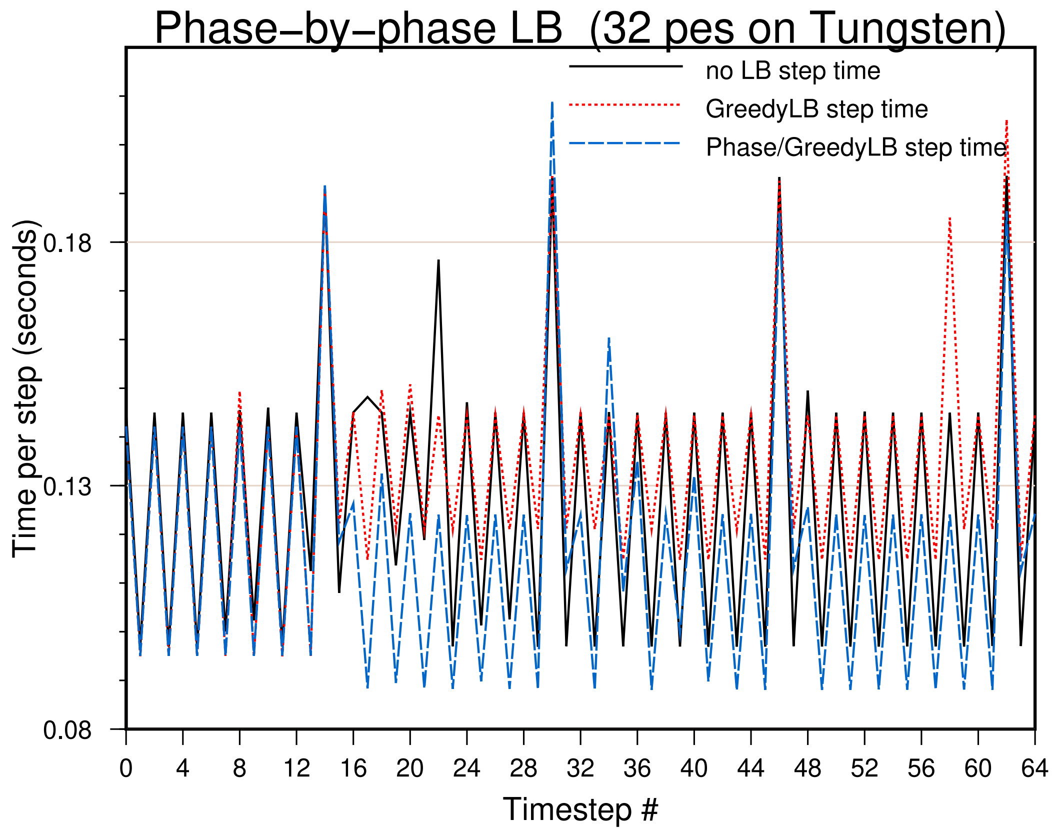
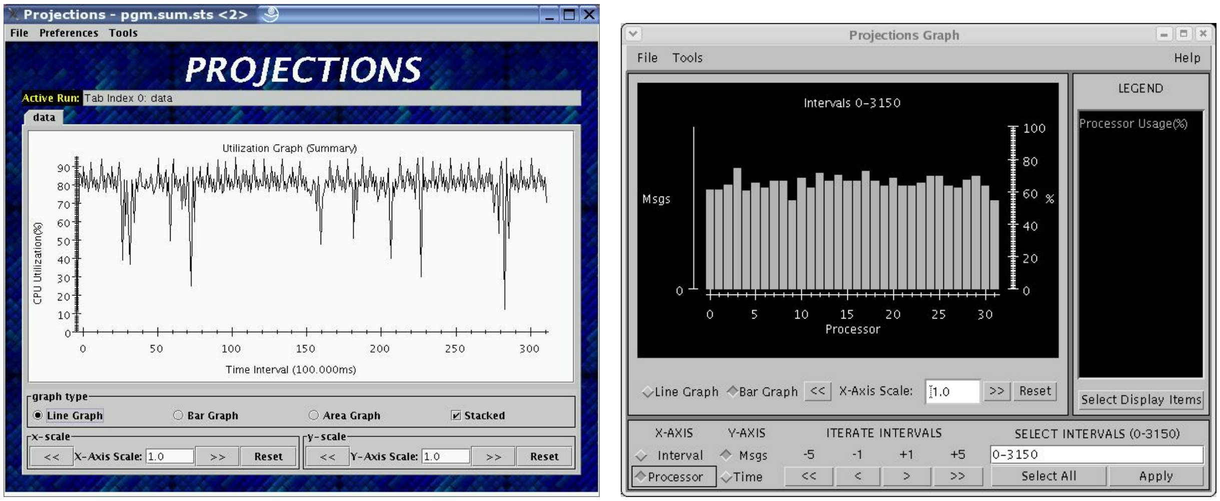


Figure 7.13: A benchmark comparison of PhaseLB and GreedyLB

processors with Greedy load balancing are shown in Figure 7.15. It is seen that although the overall CPU utilization is balanced by GreedyLB at about 60% (Figure 7.15(b)), the CPU utilization over time (Figure 7.16(a)) is worse than the result without load balancing. Finally with PhaseLB, as seen in the big dotted line in Figure 7.13 and detailed Projections view in Figure 7.16, execution time of both phases decreases, leading to the improved overall CPU utilization (around 70% in Figure 7.16(b)).

7.5 Asynchronous Load Balancing

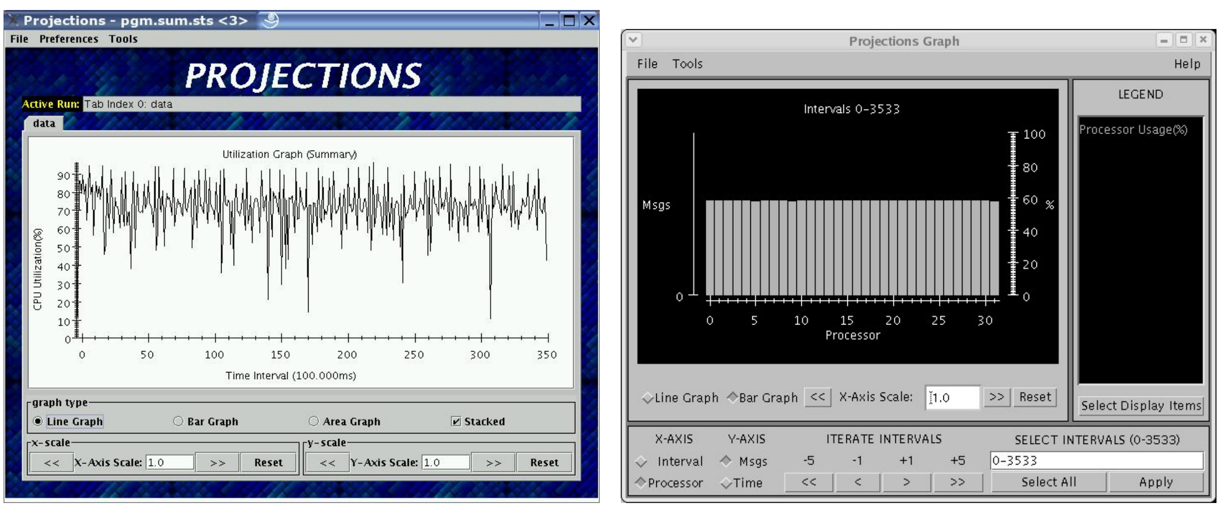
Load balancing costs may be prohibitively high when balancing the load in large scale applications, especially as in synchronous centralized load balancing schemes, as the application stops and hands over to the load balancing framework to perform load balancing until it finishes. The overhead of load balancing to the application thus includes the total time spent in load balancing, involving the computation time of load



(a) Utilization over time

(b) Utilization across processors

Figure 7.14: Projections summary view of run without load balancing



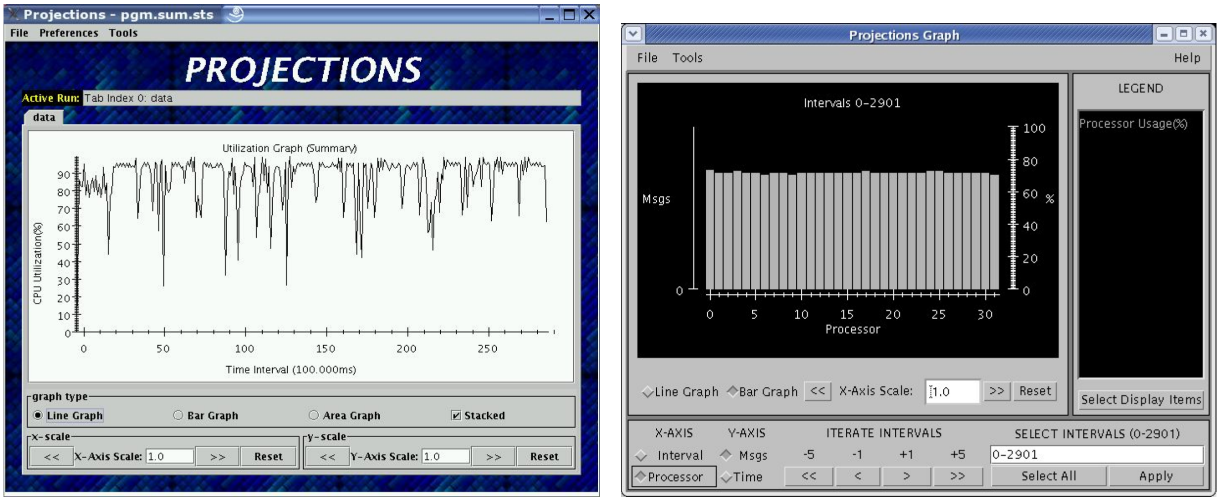
(a) Utilization over time

(b) Utilization across processors

Figure 7.15: Projections summary view of run with Greedy load balancing

balancing algorithm and the time in migrating objects. As a result, the overhead may be very high as the size of machine and/or number of migratable objects to balance increase.

There are several possible ways to control the load balancing overhead, such as optimizing load balancing algorithms to make them faster, and/or limit the number of objects to move, etc. Another possible way to *hide* the load balancing overhead is to allow overlapping of load balancing and computation, as in



(a) Utilization over time

(b) Utilization across processors

Figure 7.16: Projections summary view of run with Phase load balancing

asynchronous load balancing schemes.

In asynchronous load balancing schemes, load balancing occurs concurrently or in the background of normal execution. When a new object-to-processor mapping is calculated, the objects are instructed to migrate to their new processors. This, however, typically requires that the objects be capable of migrating *at any time*, whenever they receives the migration notification.

In the following subsections, we first discuss issues in any-time migration, followed by the details of the scheme of aynchronous load balancing.

7.5.1 Towards Any-time Migration

A classification of object types is important to the framework. Some objects may be easy to migrate at any time, but others may be difficult to migrate except at particular points in their execution. Objects which can only be migrated at particular points are *synchronized objects*. Such objects inform their array manager of potential migration points through a **ReadyLoadBalance()** [6] call, which in turn informs the LB Manager. Those objects cannot leave the migratable state until the LB manager signals the array manager to resume them. For these object arrays, at regular intervals each object calls a **ReadyLoadBalance()**, and perform no further computation until the array manager invokes their **ResumeFromLoadBalance()**

[6] Also known as AtSync().

method. Before **ResumeFromLoadBalance()** is called, the framework may or may not request that the object migrate to a new location. The **ReadyLoadBalance()** call acts as a local barrier. A local barrier is a barrier that only requires the synchronization of objects on the current processor. When all local objects call **ReadyLoadBalance()**, a load balancing cycle may be triggered.

Objects generally are not allowed to migrate voluntarily at some time between two load balancing cycles due to the use of local barrier for synchronizing the local objects. Migrating at any time may cause the local barrier to lose the head count of objects calling **ReadyLoadBalance()**. The consequence is that the local barrier might never reach the point where a load balancing cycle could be triggered.

There are a few applications, however, which require the capability of any-time migration. As an example, a phase load balancing scheme (Section 7.4) may migrate objects whenever computation phase changes, even in the middle of the computation of an object. Another example is *sub-step load balancing* (Section 7.2.3), in which case in order to tolerate a transient OS interference, an object may choose to migrate to an idle or less busy processor temporarily to accelerate the computation that may have been delayed due to the interference.

Any-time migration of a migratable object in general requires a complete checkpoint of data at any given time, including the transient state in both application and the run-time system. This is feasible but often requires tremendous programming effort. Any-time migration of a migratable thread in AMPI, however, is much more difficult due to the required migration of thread stack and heap data.

7.5.2 Asynchronous Load Balancing

The basic load balancing scheme was implemented as a "stop and go" strategy. As illustrated in Figure 7.17, when performing a load balancing cycle, processors stop and exchange load information among themselves. In case of a centralized load balancing strategy, processor 0 makes load balancing decision and broadcast the decision to every processor with a message. All processors migrate objects and then computation resumes.

This scheme may incur a large overhead due to the cost of barriers. When processor 0 is performing load balancing, all other processors have to wait until load balancing finishes on processor 0.

Asynchronous load balancing is designed to overcome this inefficiency. In this scheme, load balancing occurs asynchronously without a barrier. As illustrated in Figure 7.18, at time of load balancing, objects notify the load balancer that they are ready to migrate, but they continue with their computation. A load

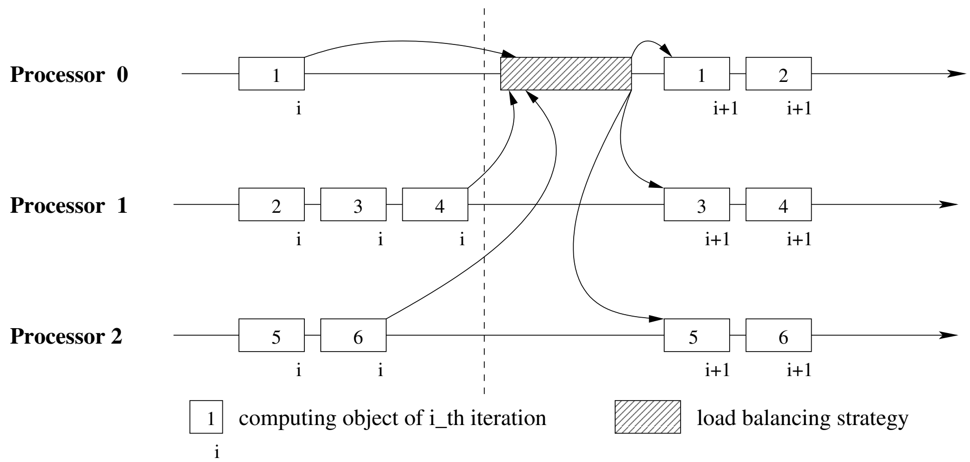


Figure 7.17: Traditional synchronous load balancing

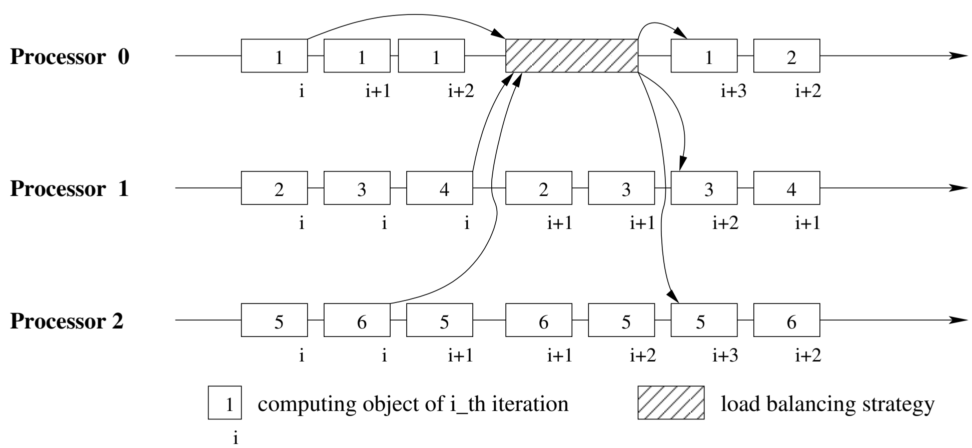


Figure 7.18: Asynchronous load balancing

balancing process then occurs simultaneously with computation. When a new load balance is calculated, a migration decision is sent to every object, which stores the destination processor to migrate. At their convenient time, these objects migrate to new processors.

There are a few advantages of asynchronous load balancing over the synchronous scheme. First, eliminating the stop barrier helps reducing the idle time on faster processors which otherwise would have to wait for the slower processors to join the barrier in order to trigger load balancing. Second, it allows the overlapping of load balancing process and computation in an application, which potentially could help improve the overall performance. Finally, object migration moment is flexible, which potentially allows overlapping of the object migration and computation. For example, an object can choose to migrate when it is going to be idle.

In applications involving objects that do not allow any-time migration, such as AMPI threads, object migration has to be restricted to particular points in their execution [7]. In asynchronous load balancing, such objects can make a call to **allow_migrate()** to notify the array manager the opportunity for migration. Work is in progress to extend the AMPI run-time to allow AMPI threads to migrate at any time.

A case study of the asynchronous load balancing scheme used in Fractography3D — a real-world AMPI application, is presented next.

7.5.3 Application Study — Fractography3D

Fractography3d is a dynamic 3D crack propagation simulation program to simulate pressure-driven crack propagation in structures. It was developed by Philippe Guebelle [8] and his students with collaboration with our group.

The fractography3d code is implemented on CHARM++ FEM framework [18]. This framework allows an application scientist interested in modeling structural properties of materials, including dynamic behavior such as crack propagation, to develop codes that embody their modeling techniques without having to pay attention to the parallelization process. FEM framework builds upon CHARM++/AMPI and its automatic load balancing framework, allowing it to automatically adapt to load imbalances resulting from the dramatic changes in computation load.

Load Imbalance

In this application, the physical domain is discretized into a finite set of tetrahedral elements. Corners of these elements are called *nodes*. In each iteration, displacements are calculated at the nodes from forces contributed by surrounding elements. Typically, the number of elements is very large, and they are grouped into a number of *chunks* distributed across processors. In each iteration of the simulation, forces on boundary nodes are communicated across chunks, then they are combined, and new displacements are calculated.

There are two factors that may contribute to the load imbalance in this application. When an external force is applied to the material in study, the initial elastic state of the material may change into plastic along the wave propagation, which results in much heavier computation. Second, to detect a crack in the domain, more elements are inserted between some elements depending upon the forces exerted on the nodes. These

[7] AMPI threads normally can only migrate when there is no pending receives.

[8] Philippe Guebelle is a professor of Dept. of Aerospace Engineering at University of Illinois at Urbana-Champaign

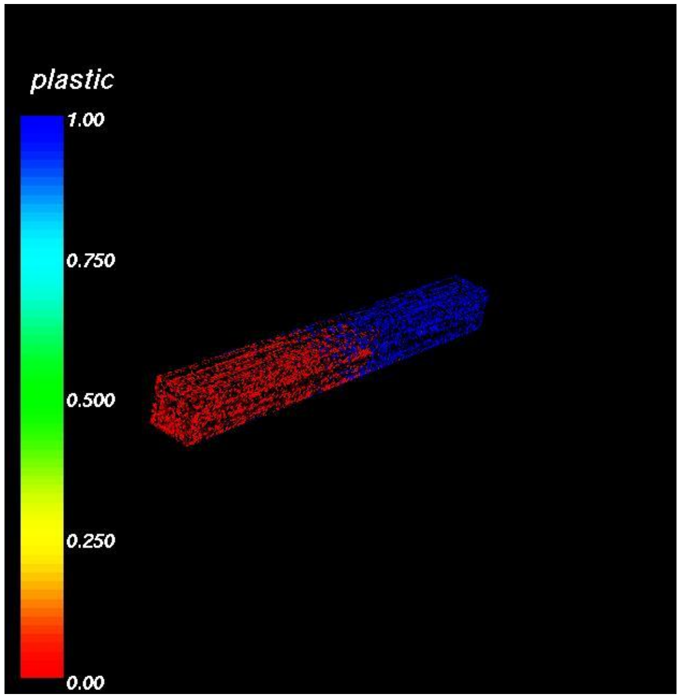


Figure 7.19: Elastic bar turning into plastic with 1D force

added elements, which have zero volume, are called *cohesive elements*. At each iteration of the simulation, pressure exerted upon the plastic structure may propagate cracks, and therefore more cohesive elements may have to be inserted. Thus, the amount of computation for some chunks may increase during the simulation. This results in severe load imbalance.

Simulation with Fast Changing Load Imbalance

To evaluate the performance of load balancing and test the new asynchronous load balancing scheme (Section 7.5), we first examined a synthetic problem with an elastic bar, as shown in Figure 7.19. In this simulation, a force is exerted from the front of an elastic bar, the wave of force propagates to the back of the bar and is bounced back. During this process, the elastic bar turns into plastic state along with the wave of force.

The test was run on 32 processors of the SGI Altix at NCSA and used 160 AMPI virtual processors. Figure 7.20 shows the result without load balancing in a Projections utilization graph over time interval. The total execution time without load balancing was 207 seconds. As the figure shows, in the first few seconds the utilization is low because of the initialization task, where the code builds a graph from an input file and

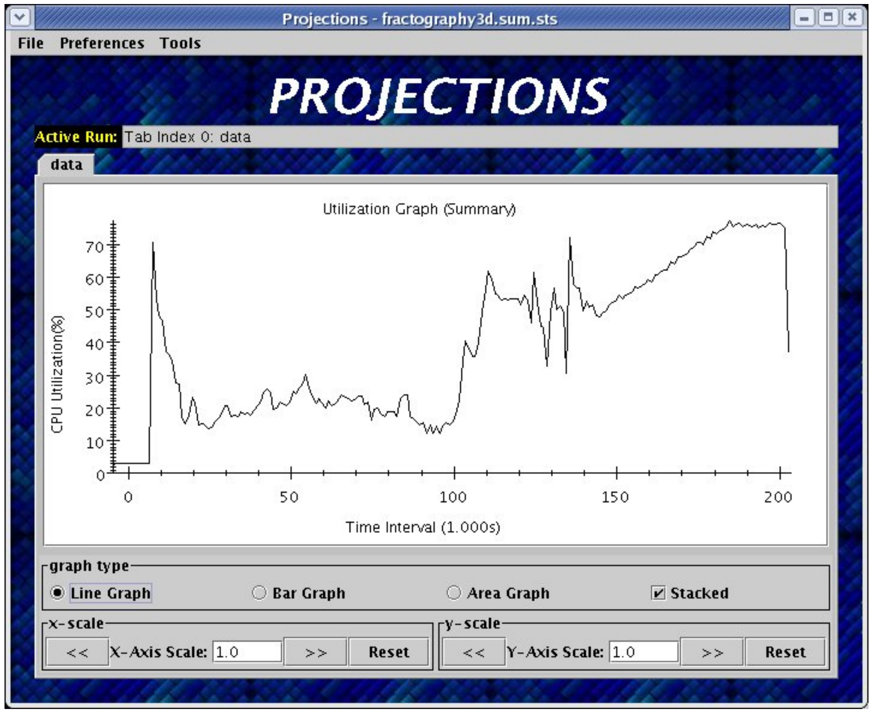


Figure 7.20: CPU utilization projections graph of Fractography3D without load balancing

calls METIS partitioner to partition the graph into meshes.

In the second run, we ran the same test with GreedyLB. The result is shown in Figure 7.21 in the same Projections utilization graph. The load balancing is invoked around time interval 100 in the figure. After the load balancing, the CPU utilization is slightly improved, with improved total execution time of around 198 seconds.

Finally, we ran the same test with the GreedyLB in an asynchronous load balancing scheme. Asynchronous load balancing avoids the stall of an application for load balancing and overlaps the execution with the load balancing and migration. The result is shown in Figure 7.22 in the Projections utilization graph. It can be seen that after load balancing, the overall CPU utilization was further improved, and the total execution time is improved to 187 seconds, which is a 20 second improvement in the final period of the execution (about 100 seconds) after load balancing occurred.

Larger Simulation

Given the promising result with load balancing, we then moved forward to a real-world test problem with a crack propagation simulation. This test simulates a crack propagation with a force and the process of elastic

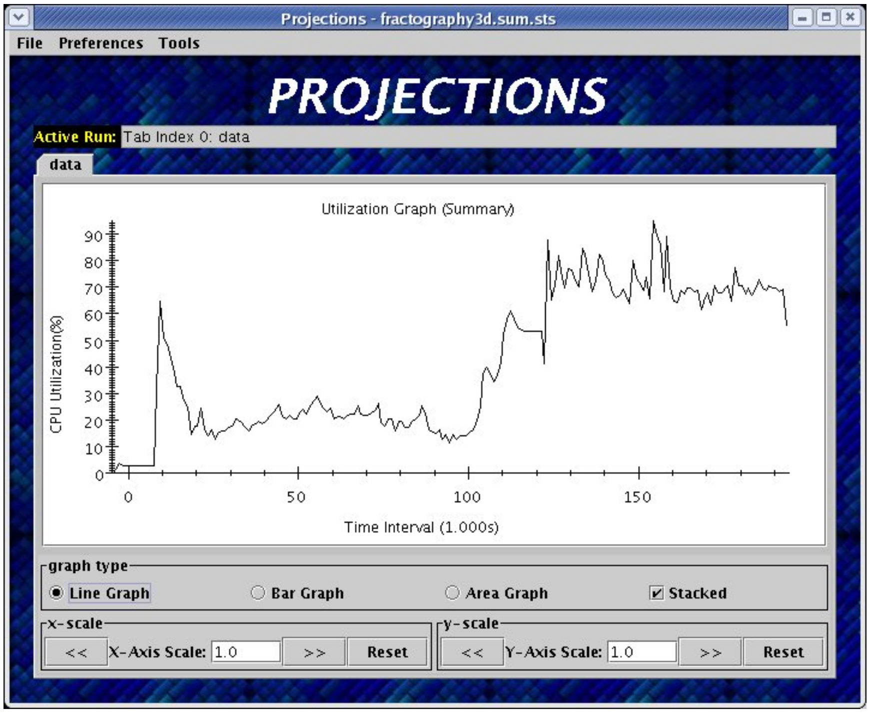


Figure 7.21: CPU utilization projections graph of Fractography3D with synchronous load balancing

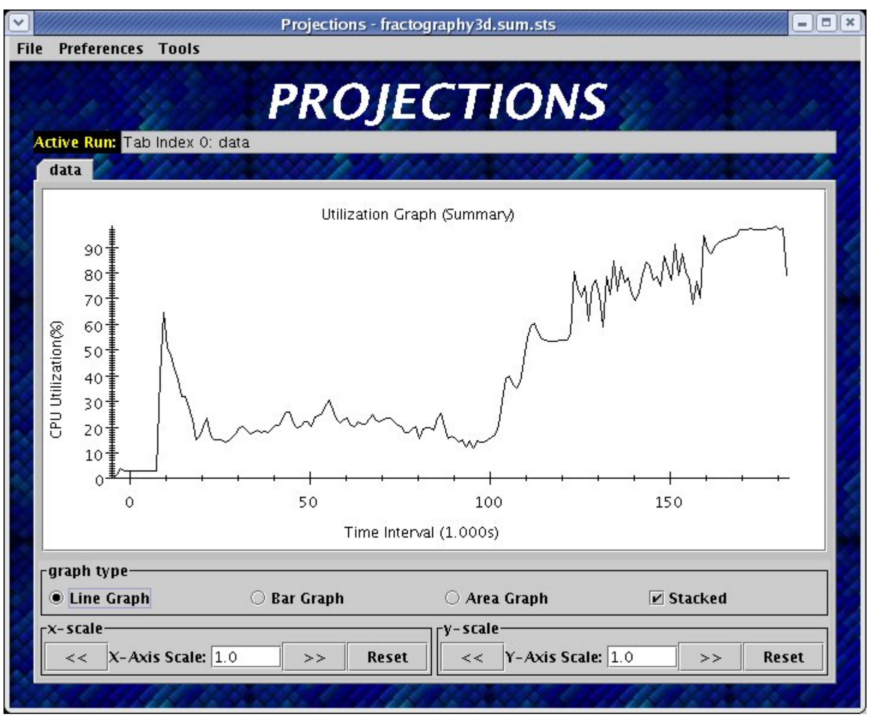


Figure 7.22: CPU utilization projections graph of Fractography3D with asynchronous load balancing

turning into plastic zone along the crack, as illustrated in Figure 7.23. The crack propagation simulation was run with 1000 AMPI virtual processors on 100 processors of the Turing Apple cluster at UIUC.

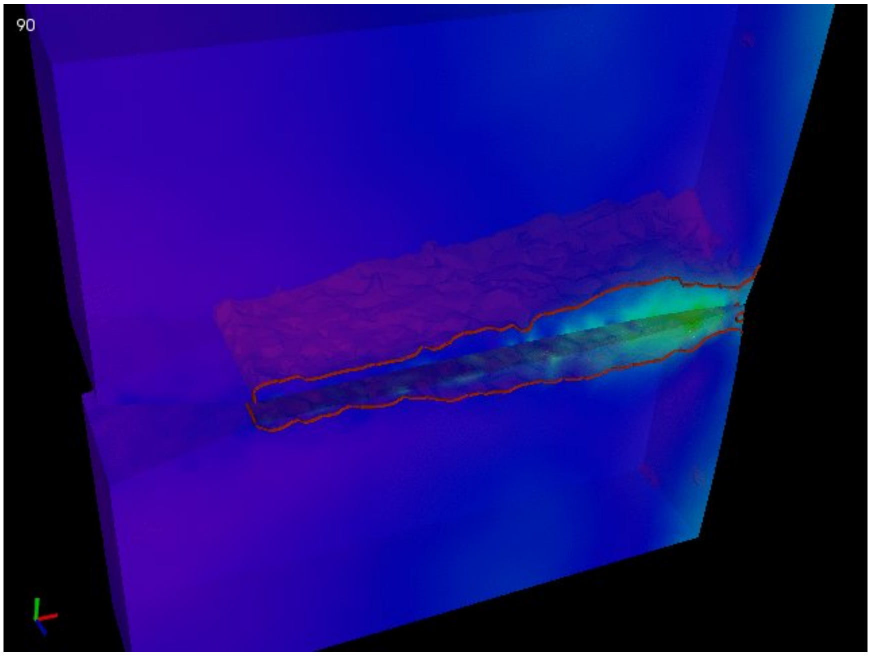
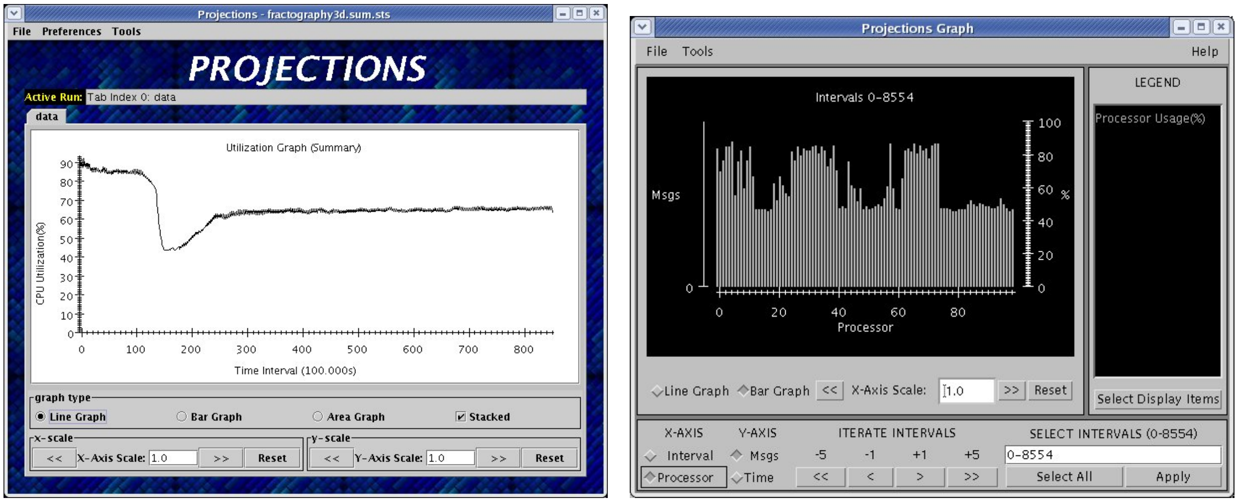


Figure 7.23: 3-D elastic-plastic fracture

The simulation without load balancing runs for 24 hours. The Projections view of CPU utilization over time interval is shown in Figure 7.24(a). It can be seen that around time interval 120, the CPU utilization dropped from around 85% to only about 42%. This is due to the start of the process of elastic turning into plastic zone along the crack, leading to load imbalance. As more elastic parts turn into plastic, the CPU utilization slowly increases until all turn into plastic. The load imbalance can also be easily seen in the CPU utilization graph over processors in Figure 7.24(b). While some of the processors have the CPU utilization as high as about 90%, some processors only have about 50% of the CPU utilization during the whole execution.

Figure 7.25(a) presents results of automatic load balancing of the same crack propagation simulation in the view of overall CPU utilization over the time intervals. The load balancing is invoked every 500 time-steps of the simulation, with a greedy-based algorithm. The automatic load balancer uses the runtime load and communication information instrumented by the CHARM++ runtime to migrate chunks from the overloaded processor to underloaded ones, leading to improved performance. As figure 7.25(a) shows, the overall CPU utilization on all processors throughout the entire simulation stays around 80-90%. Fig-

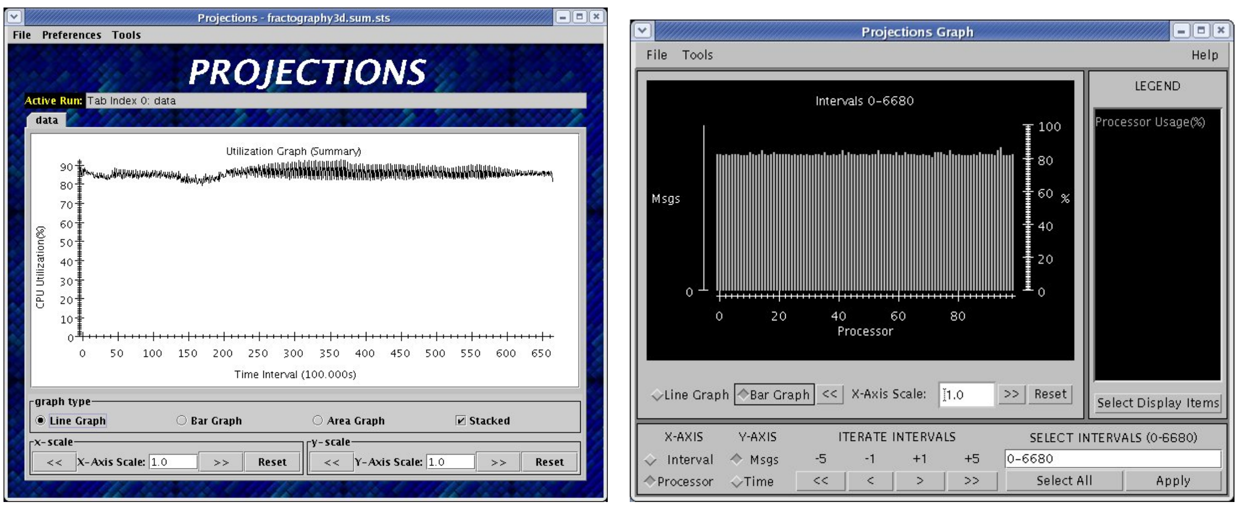


(a) CPU utilization over time intervals

(b) CPU utilization across processors

Figure 7.24: CPU utilization projections graph of Fractography3D without load balancing

ure 7.25(b) further illustrates that load balance has been improved from figure 7.24(b) in the view of the CPU utilization over processors. It can be seen that a CPU utilization of at least 80% is achieved on all processors with negligible load variance. The simulation with load balancing now takes about 18.5 hours to complete, yielding about 23% of performance improvement from load balancing.



(a) CPU utilization over time intervals

(b) CPU utilization across processors

Figure 7.25: CPU utilization projections graph of Fractography3D with load balancing

Chapter 8

Load Balancing Strategies for Peta-scale Machines

Load balancing on peta-scale machines is a highly challenging problem due to the large scale of the parallel system and application. In the context of CHARM++, in order to exploit such peta-scale machines and take advantage of the load balancing capability of CHARM++, applications typically involve far more large number of migratable objects than the system size. The complexity of load balancing algorithms thus leads to significant overhead of load balancing. For instance, consider a load balancing algorithm of complexity $O(NlogN)$ where N is the number of migratable objects. When the number of objects is increased from $4K$ to $64K$, the load balancing decision making time increased by 48 times, which leads to considerable overhead that makes load balancing less useful.

Although a lot of work has been done in designing load balancing strategies for large systems consisting of a few thousands of nodes, very few strategies in literatures address load balancing issues for very large parallel machines with tens of thousands of nodes efficiently.

In this chapter, we first summarize the challenges involved in load balancing for peta-scale machines. We then present studies of the scalability and limitations of the traditional load balancing methods including the fully distributed and centralized load balancing strategies. Finally, we present a new hybrid load balancing algorithm to tackle the problem. We demonstrate the performance of the new load balancing method on peta-scale machines using the simulator we have developed.

8.1 Load Balancing Challenges for Peta-scale Machines

Load balancing on a very large number of processors is more challenging compared to conventional parallel machines of a much smaller size. For very large parallel machines, the load balancing problem needs to face these new challenges:

1. Machine topology becomes an important factor that load balancing strategies need to take into account. In most conventional schemes used for smaller parallel machines, the machine topology often is ignored by assuming that message latency between any pair of processors in the system is nearly the same. By ignoring topology information, a load balancer is allowed to make load balancing decisions in a way that potentially can place two intensively communicating migratable objects on two distant processors. For peta-scale machines, due to the extremely large number of processors, although latency is still low, the fraction of bandwidth used by communicating objects increases with the number of hops that messages traverse. Thus, topology can no longer be ignored.

2. It is very difficult to make well-informed load balancing decisions without some global load information about the system, especially when the parallel system is very large. However, it may be very expensive or even impossible due to memory limits to collect such global state including both computation and communication data. Thus, the load balancing overhead can no longer be a secondary consideration in this case. When the cost of load balancing significantly outweighs the possible benefits of the balanced load, load balancing is then prohibitively expensive.

3. Load balancing algorithms tend to be much slower when the number of processors and/or the number of migratable objects grow. As a simple example, for an application with N objects running on P processors, with a load balancing algorithm of $O(Nlog(P))$ complexity, when N increases from 16K to 1M and P increases from 1K to 64K [1], the execution time of the algorithm will be 25.6 times larger.

Compared to centralized strategies, a fully distributed load balancing strategy is designed to be scalable on very large machines. In the next section, we will study the load balancing techniques in distributed strategies and their performance constraints.

[1] This is to keep the same degree of virtualization.

8.2 Fully Distributed Load Balancing Strategies

Distributed load balancing, also called diffusive load balancing in some literature, is a decentralized and scalable implementation of a load balancing strategy for a massively parallel machine. It is based on the generic physical principle of diffusion in which energy or matter flows from higher concentrations to lower concentrations, leading to a homogeneous distribution.

Distributed load balancing strategies are based on replicated decision components, each with the same behavior and capable of autonomous and synchronous activity. The load balancing goal is locally pursued: the scope of the actions for each decision is bound to a local area of the system.

8.2.1 Measurement-based Neighborhood Averaging Strategy

The measurement-based *Neighborhood Averaging Strategy* was implemented in Charm++. In this scheme, instead of having a central controlling node for collection of the object-communication graph and making load balancing decisions, each processor sends summary statistics about its load to *neighboring processors*. Each processor (the central processor) and its neighboring processors form a load balancing domain. Neighbor nodes are determined using either the physical or a virtual topology of processors such as ring, torus or dense graph. The central processor performs load balancing based on the average load in the domain. If the load of the central processor is below the neighborhood average, it does not need to continue the load balancing task. Only an overloaded central processor makes load balancing decisions to migrate objects away to its underloaded neighboring processors.

8.2.2 Neighborhood Averaging with Work-Stealing when Idle

Seed load balancing is another type of fully distributed load balancing strategy implemented in Charm++. It treats the load balancing problem as a task scheduling problem (Section 6.3). In Charm++, tasks are initially represented by object creation messages, or "seeds". Seed load balancing involves the movement of seeds, to balance work across processors.

Most of the work in the seed load balancing problem was done in early 90's including Adaptive Contracting Within Neighborhood (ACWN) [46, 80] and a prioritized load balancing strategy [83].

Although the methods discussed above are well studied, we found several problems in practice. For example, seed load balancing schemes suffer from the aging of load information. This is largely due to the non-preemptive message scheduler implemented in Charm++ — when a message is being executed, other messages have to wait in the message queue. Thus the processing of critical messages that contain the load information may be delayed in the queue and get out-of-date when the execution time of the method being processed is long. This aging of load information may lead the load balancing runtime to make poor load balancing decisions. Also, the delay in processing the load messages may further delay the invocation of the load balancing strategies. This is because seed load balancing can only be triggered when all load information is received from neighboring processors. Any delay in receiving these messages may slow down the invocation of the load balancing strategies. In applications with abrupt changes of load, this may lead to poor performance.

A possible solution provided in PREMA [13] is to use a multi-threaded approach. As proposed in the PREMA paper, a "polling thread" is spawned whenever a long-running work unit is executing. This thread polls the network for load balancing messages for a period of time, allowing each processor to maintain up-to-date load information, as well as satisfying load balancing requests in a timely fashion. Once the work unit finishes execution, the polling thread is killed and only the application's main thread remains. This approach, however, involves high overhead associated with the polling thread such as time-slicing overheads. The creation and killing of the polling thread also requires the long-running work unit to be "long" enough to offset the thread overhead in order to produce any advantage. This scheme also requires the run-time system to accurately predict the running time of each work unit to decide whether a polling thread needs to be spawned, which may be difficult.

In Charm++, we tackle this problem with a different approach. We designed and implemented "immediate messages" — a message that can interrupt the running work, which, when combined with scheme for work-stealing at idle time, provides an efficient solution.

Immediate Messages

Immediate messages are "short" messages associated with handler functions that can be invoked "immediately" when the message arrives. The execution of immediate message handlers may interrupt any execution of any handler of a non-immediate message; therefore immediate messages provide an efficient mechanism for timely processing of messages. Immediate messages can be useful, for example, in implementing tree-based multicasts/reductions operations, in which processing of a critical message along the tree is not delayed in the scheduler queue by some long-running work.

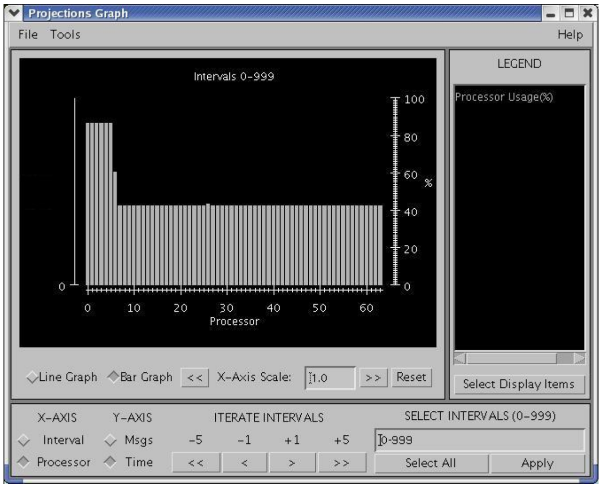
To guarantee the responsiveness to an immediate message, a few approaches are implemented. For low level machine specific layers that use UDP sockets for communication, sockets are set in asynchronous I/O mode and an O.S. interrupt is issued whenever an incoming message arrives on a socket. The immediate message handler can then be executed right inside the signal handler that interrupts the current running work. For machine layers that implement SMP mode with multi-threaded worker threads with an additional communication thread, an immediate message handler can be executed immediately by the communication thread when the message is received, which then overlaps with execution of handlers of other messages on the worker threads.

In seed load balancing, immediate messages are used in two cases. In the first case, it is used when a processor sends the load information message to its neighboring processors. The neighboring processors receive the load information and are able to process the information immediately to decide whether to initiate a new load balancing. In the second case, immediate messages are used in *work-stealing*, which works as follows. When a processor becomes idle, it sends a *work request* in an immediate message to its neighboring processors "stealing" their work. When neighboring processors see the request, in the immediate message, they can promptly respond by sending some work to the idle processor regardless of what is currently being executed.

Experimental Results

We used a benchmark similar to the test presented in [14] to evaluate our new seed load balancer. This benchmark creates 80,000 objects, 10% of which are "heavy" and others are "light". The heavy objects have twice as much work as the light ones. To stress the load balancers, heavy objects are initially assigned to a subset with the lower 10% of the processors.

Figure 8.1 illustrates the results obtained on 64 processors of a cluster without and with a seed load balancer. Figure 8.1(a) shows the processor utilization over processors for the run without a seed load balancer. As it clearly shows, the first 10% of processors have twice as heavy a workload as the rest, due to the initial mapping. The completion time without load balancing was 99.85 seconds on the 64 processors. Figure 8.1(b) shows the processor utilization for the run with the new seed load balancer. The average processor utilization during the run time was improved to about 93%. The completion time was reduced to



(a) Percentage of processor utilization of the benchmark without seed load balancing. 10% of the 80,000 objects are twice as heavy as the rest. The heavier objects are initially assigned to first 10% processors

(b) Percentage of processor utilization of the same benchmark with seed load balancing turned on.

Figure 8.1: Seed load balancing benchmark test

55.52 seconds, yielding about a 44% improvement in performance.

8.2.3 Limitations of Fully Distributed Strategies

Although fully distributed load balancers are designed to be scalable for very large machines, they are likely to suffer from the following performance issues:

1. In a fully distributed scheme, each node makes autonomous decisions, and no node has a global vision of the state of the entire system. Thus, an optimal load balancing solution is hard to achieve, especially in a rapidly changing environment. Occasionally, load balancing may make wrong decisions and the application can enter into a state when tasks keep migrating without getting settled down on one node (so called "task thrashing").

2. Fully distributed strategies use only a small amount of information about the state of the system; typically one node only communicates with a limited number of neighboring nodes for exchanging load information. If the most heavily and the most lightly loaded sections of processors happen to be a large distance apart (which is likely to happen when the machine topology diameter is big), a fully distributed scheme with limited amount of information available to a node may take a number

of iterations to propagate the load between the two ends. The situation may become even worse if the computation load changes rapidly, leading to very poor performance.

The following case study with NAMD further illustrates the performance limitations of a fully distributed load balancing strategy.

8.2.4 NAMD Performance with a Fully Distributed Strategy

We studied an *Neighborhood Averaging Strategy*, a fully distributed load balancing strategy described earlier, in the context of NAMD program.

The test was run with Apo-A1 benchmark on 256 processors of LeMieux at PSC. The simulation ran for a total of 500 timesteps. The distributed load balancing was invoked once every 20 timesteps, corresponding to a total of 25 load balancing steps (as shown as the top curve in Figure 8.2). We also compared its performance with a greedy centralized load balancing, as shown in the bottom curve of Figure 8.2. In the run with centralized load balancing strategy, the load balancing was invoked only once.

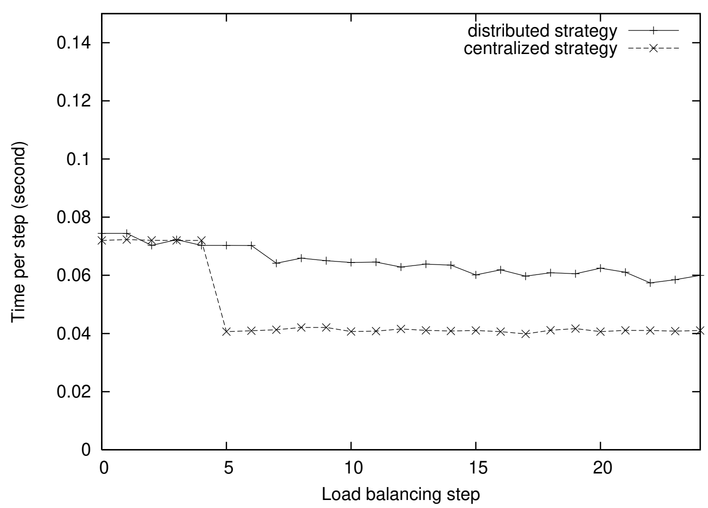


Figure 8.2: NAMD performance comparison between a distributed LB strategy and a centralized LB strategy

In NAMD, initially the migratable objects are not mapped to processors very carefully. Thus, the simulation may start with significant load imbalance. As shown in the figure, the centralized load balancing strategy was able to improve the load balance significantly and obtained much better performance, and the performance sustained due to the slow changes in load. In comparison, the distributed load balancing strategy did not perform as well as the centralized strategy. It can be seen that although the distributed load balancing strategy improved the simulation performance, this tends to occur in a much slower fashion. During the total of 500 timestep simulation, distributed load balancing failed to achieve the simulation speed of the centralized load balancing strategy.

8.3 Centralized Load Balancing Strategies for Peta-scale Machines

Centralized load balancing strategies have the advantage that they normally have access to global load and communication information of the entire system, which enables them to yield better load balancing decisions. However, centralized strategies are inherently non-scalable. They have the following limitations when applied to very large systems.

1. Collection of global information may become a prohibitively expensive process for a very large system.
2. Global information is collected to one central node to make the load balancing decisions. The memory usage for storing the global state information on the central node can be prohibitively high.
3. The single central node can become the communication bottleneck in a parallel system with a very large number of processors.
4. The execution overhead of a centralized strategy's decision-making algorithm can be very high, given the large number of processors and migratable objects.
5. Greedy-based centralized strategies tend to make global decisions without considering the existing object mapping; they tend to lead to migrating almost all objects away from their current location which can be very expensive in a very large system.

In next section, we will further illustrate and quantify these limitations of centralized strategies on machines with as many as 64K processors using the BigSim simulator we developed.

8.3.1 Simulation Study of Centralized Strategies

Intuitively, the overhead of centralized strategies is known to be very high. To demonstrate the overhead quantitatively in a more realistic environment, in this section, we studied the overhead of several existing Charm++ centralized load balancers on very large machines (up to 64K processors) using the BigSim simulator we developed (Chapter 4).

Benchmark

The **lb_test** benchmark is a parameterized program that creates a specified number of migratable objects communicating in a 2-D mesh topology. The work done by each objects in each iteration is randomized in a parameterized range.

We ran the load balancing benchmark lb_test on the BigSim simulator to simulate centralized load balancing on a BlueGene/L like machine with 32K and 64K processors using only 32 processors (8 nodes) of PSC LeMieux to simulate. In our tests, we studied the load balancing overhead including memory usage and time costs by varying the number of migratable objects from 128K to 1M.

Memory Usage

We measured the memory usage on the central node for various experimental configurations. The results are shown in Table 8.1. The memory usage reported is the total memory needed for storing the object-communication graph on the central node. The peak memory usage of the load balancing algorithm itself is not included.

	Number of Objects			
num of pes	128k	256k	512k	1m
32k	59M	115M	228M	455M
64k	61M	117M	230M	457M

Table 8.1: Memory usage of centralized load balancers on the central node

As the result shows, the memory overhead of the load “database” used for a centralized load balancing strategy increases significantly as the number of objects increases. For applications with 1 million objects running on 64K processors, the database alone requires around 450MB of memory, which will barely fit on a node of a BlueGene/L (which has 512MB total memory); however, the time overhead may also be large as shown in the next section.

Execution Time in Load Balancing Algorithms

We simulated the lb_test program with varying number of migratable objects on machines with 32K and 64K processors. We stored the load databases on the central node on to a disk of LeMieux, and transferred them to a Linux machine in our lab. We then experimented with our sequential load balancing simulation tool (Section 7.1.1) on the Linux machine with the database file to measure the execution time of a few Charm++ load balancing strategies. The machine that ran the sequential simulation has an Intel Pentium 4 CPU at 2.8GHz.

The results of the simulation on 32K processors are shown in Table 8.2, and the results for 64K processors are shown in Table 8.3.

	Number of Objects			
Strategies	128k	256k	512k	1m
GreedyLB	0.517	1.103	2.280	4.756
GreedyCommLB	12.350	25.867	59.334	119.919

Table 8.2: Load balancing strategy time in second for 32K processors (on finesse)

	Number of Objects			
Strategies	128k	256k	512k	1m
GreedyLB	0.521	1.111	2.259	4.809
RefineLB	29.412	36.720	80.567	94.237
RefineKLB	9.269	43.678	-	-
GreedyCommLB	41.289	94.75	117.594	386.02

Table 8.3: Load balancing strategy time in second for 64K processors (on finesse)

As the results demonstrate, the execution time of some load balancing algorithms may be very long (the longest was over 6 minutes). Further, the execution time of the tested load balancing algorithms varies significantly. To our surprise, the algorithm in GreedyLB takes only a few seconds for balancing 1 million objects on a machine with 64K processors. In general, load balancing strategies that do not take communication into account are much faster than the ones that take communication into account (such as GreedyCommLB).

Total Load Balancing Execution Time

In this test, we simulated the complete process of two typical centralized load balancing strategy GreedyLB and GreedyCommLB on a machine with 64K simulated processors with a varying number of migratable objects. The total load balancing execution time includes the time for load statistics collection, strategy execution time and the time for object migration. The results are shown in Table 8.4.

	Number of Objects		
Strategy	256k	512k	1m
GreedyComm	100.59	209.00	418.9
Greedy	7.60	15.56	31.66

Table 8.4: Total Load balancing execution time for 64K processors (in seconds)

It can be seen that the total overhead of the GreedyCommLB (shown in the first row) is significant in this test. The load balancing overhead can be as long as 7 minutes for an application of 1 million objects on 64K processors. With a synchronous load balancing scheme, this means a 7 minute pause of the application for load balancing, which is prohibitively expensive. Compared to GreedyCommLB, the load balancing execution time for GreedyLB is much shorter. For the run with 1 million objects on 64K processors, it only took about 31 seconds in total, for smaller application with only 256K objects, the load balancing time reduced to only around 7 seconds.

Table 8.5 shows the maximum predicted load after load balancing, a performance metric that represents the quality of the load balancing strategy outcome (Section 7.1). We will later use these results in a comparison with a newly proposed load balancing strategy.

	Number of Objects		
Strategy	256k	512k	1m
GreedyComm	0.0279/0.0264	0.0539/.0527	0.1070/0.1056

Table 8.5: Maximum predicted load over average load after load balancing for 64K processors

8.3.2 Summary

We have demonstrated that the overhead associated with centralized load balancing strategies is significantly high. This clearly limits their usefulness on very large parallel machines, especially for applications that require frequent and/or rapid load balancing.

However, there are a few cases where centralized load balancing strategies may still be useful on fairly large parallel machines. Since the usefulness of centralized strategies is largely limited by the amount of resource required on the central node, as long as that resource allows, it may still be applicable. Furthermore, applications with slow changes in load may just need only a few load balancing steps. Therefore, slightly expensive centralized load balancing strategies may still be acceptable if the improvement of the performance in the subsequent execution outweighs the load balancing overhead.

A few optimizations on centralized load balancing scheme can further improve the performance of the centralized load balancing on large parallel machines. For example, to maximize available memory capacity for the central node, a separate node can be reserved solely for load balancing purposes, which is then free of application memory usage. Reserving one dedicated node for load balancing out of thousands of total nodes may be affordable. Blue Gene/L has a separate tree-based communication ports for broadcasting. Load balancing can take advantage of this specialized network for the statistics collection and the broadcasting of the load balancing decisions. In general, load balancing can take advantage of dedicated resources to reduce its own overhead.

8.4 Hybrid Load Balancing Strategy

We have demonstrated that both existing centralized and fully distributed load balancing strategies do not work well or achieve satisfactory result in the context of peta-scale machines. It is clear that a hybrid strategy that combines both the centralized and fully distributed strategies is needed for load balancing very large parallel machines.

Although there have been a number of such hybrid strategies studied in literature (Section 8.5), most of them are designed for applications that assume a continuous flow of tasks (e.g. state space search) which are not suitable for most scientific applications we are targeting which have persistent computation that are iterative in nature. Given the size of the machine and problem size these applications deal with, it is very expensive in terms of both time and memory for load balancing strategy to take the communication pattern into account in minimizing network communication. In this section, we propose and evaluate a new hybrid load balancing algorithm (HybridLB) that is designed for scientific applications with persistent computation and communication pattern. The proposed new algorithm takes advantage of CHARM++'s virtualization idea for applications with fine-grained parallelism, and it takes communication into account for achieving satisfactory load balancing decisions.

The basic idea in our hybrid strategy is similar to those traditional hierarchical load balancing strategies. It divides the processors into independent autonomous sets of groups and the groups are organized in hierarchies, therefore decentralizing the load balancing task. For example, a binary-tree hierarchical organization of an 8-processor system with hypercube network is illustrated in Figure 8.3. In the figure, groups are organized in three hierarchies — at each level, a root node of the sub-tree and all its children nodes form a load balancing group.

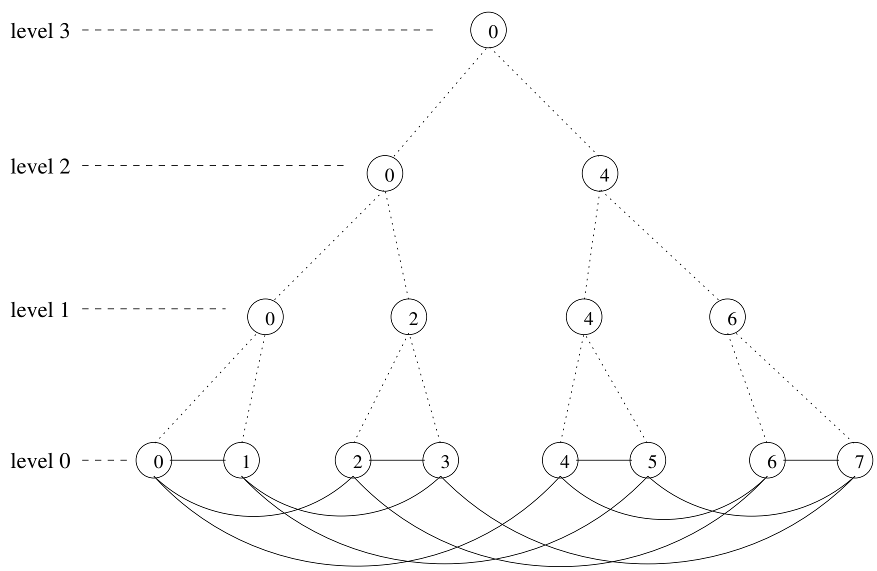


Figure 8.3: Hierarchical organization of a 8-processor system with hypercube network

In our hybrid load balancing strategy, an intermediate node at level l_i and its immediate children nodes at level l_{i-1} form a load balancing domain, with the root node as group leader. Load balancing group leaders control the load balancing process inside their domain, playing the role similar to the central node in a centralized load balancing scheme. Root processors at level l_i also participate the load balancing process controlled by their group leaders at level l_{i+1}, while those processors in the subtree of level l_i do not participate the load balancing process at level l_{i+1}. Thus the size of a load balancing domain (i.e. the number of processors) is same across all levels if the tree is uniform.

Optimization for reducing communication overhead in load balancing process itself is critical considering the size of a peta-scale machine. In our algorithm, communication for collecting application time and communication pattern is only limited in processors of a load balancing domain at each level. When load balancing occurs at level l_{i+1}, all participating processors at level l_i act as representatives of their respective processors in the sub-trees. They don't talk directly to the processors in the subtrees.

The hierarchical tree used in our hybrid load balancing strategy can be built in very flexible ways to take advantage of the architecture. A hierarchical tree can be built according to the machine's topological hierarchy to minimize load balancing communication overhead. Our experiments demonstrate that the depth of the tree is a tradeoff between load balancing cost and quality. Intuitively, trees of higher depths tend to involve less load balancing cost due to smaller sized load balancing domain at each level, but may not produce as good load balancing decisions as those trees with smaller depths. Section 8.4.1 discusses a heuristic to build such a load balancing tree.

The use of a hierarchical tree determines that load balancing cost at different levels of a hierarchical tree is not uniform - the higher level a load balancing domain, the bigger overhead it involves at that level. Unlike some other hierarchical strategies that apply same load balancing algorithm across all load balancing domains at different levels of a hierarchical tree, our hybrid strategy adopts adaptive load balancing strategies at different levels. The adaptive load balancing strategies have several advantages. First, it helps to control the memory usage and CPU overhead in executing a load balancing algorithm when the load balancing overhead increases dramatically as load balancing level increases. A detailed optimization algorithm may incur prohibitive overhead in load balancing at top levels of the hierarchical tree. This motivates us to apply different load balancing algorithms of different complexity for load balancing at different levels of a hierarchical tree. Second, our adaptive load balancing strategy can help maximizing the benefits of load balancing and avoid global work redistribution. It can take advantage of the rich set of the aggressive and refinement-based centralized load balancing algorithms we have developed (described in Section 7).

8.4.1 HybridLB Strategy Details

We describe the HybridLB load balancing strategy in the four distinct load balancing steps : load evaluation, load balance initiation, load balance decision making and object migration (Section 6.4).

Load Evaluation

During execution, CHARM++ run-time measures the time and communication pattern and store them in local database on each processor. The time data is stored in the database as a per-object data structure

$LDObjData$, while the communication data is stored in a per communication-link data structure called $LDCommData$.

Load balancing domains periodically exchange the load databases of their processors. This process is triggered by leaf processors at level 0 of the tree, who starts to send their local load database up to the domain group leader processors which are at level 1. The same process continues by ascending the tree to the top level in the tree.

Algorithm 3: Load database integration algorithm for a load balancing domain leader

```
begin
    LDObjData objdata[];
    LDCommData commdata[];
    int objcount = 0;
    int commcount = 0;
    int mem = 0;
    for i ← 0 to nChildren − 1 do
        LDStats data;
        Receive data from one child;
        for j ← 0 to data.n_objs do
            data.objdata[j].onPE = thisPE;
            objdata[objcount + +] ⟵ data.objdata[j];
        for j ← 0 to data.n_comm do
            data.objdata[j].sender = thisPE;
            commdata[commcount + +] ⟵ data.commdata[j];
        mem+ =
        data.n_objs ∗ sizeof(LDObjData) + data.n_comm ∗ sizeof(LDCommData);

    if parentPE! = −1 then
        send loaddata and commdata to parentPE;
end
```

The algorithm is described in Algorithm 3. At each level l_i, each load balancing domain leader receives load statistics from children at level l_{i-1}. The received load data is integrated into its own load database. Load data is preprocessed so that it looks as if all objects belonged to the domain leader, and all communication senders were the domain leader (two inner for loops in Algorithm 3).

As load information propagates to the higher level, the amount of load data increases dramatically since the total number of processors covered in a load balancing domain expands when level increases. This greatly impacts the memory capacity needed to store the integrated load database on a node. Therefore, it is necessary to shrink load data while propagating to the higher levels. Hybrid load balancer uses a memory usage estimator to monitor the memory usage due to the load data. Given a limit to the memory usage $\bar{M}$ on a node [2], the amount of memory needed for storing load database can be easily calculated as:

$$m = n\_objs * sizeof(LDObjData) + n\_comm * sizeof(LDCommData)$$

, where n_objs is total number of objects, n_comm is the number of communication records, *LDObjData* is the data structure recording the per object load data and *LDCommData* is the data structure recording the per communication data. When $m > \bar{M}$, data needs to be shrunk to fit in memory. Several strategies are developed in HybridLB:

- At each level, communication data can be shrunk by deleting some of the trivial communication records between objects. The heuristics is that it is more important for load balancing algorithm to optimize communication of the most communicating objects. This scheme does not involve much loss in load data.

- When the amount of load data is prohibitively large at a certain level, a dramatic shrinking scheme is performed. In this case, only the total CPU load information is sent to the higher level and load balancing at this domain switches to a different mode — "semi-centralized load balancing" which is described in the following section.

Load Balance Initiation

Each load balancing domain performs load balancing independently when load data is updated. The load balancing starts in a top-down fashion in the hierarchical tree. That is, the root of the hierarchical tree, which has the access to the global load information, starts the load balancing.

In each load balancing domain, load imbalance is computed as: $\frac{L_{max}}{L_{avg}} - 1$, where L_{max} is the maximum load of nodes in the load balancing domain, and L_{avg} is the average load of all the nodes in that domain.

When load imbalance in a domain is greater than a threshold (e.g. 5%), load balancing is invoked in this domain with the designated load balancing strategy at this level. Load balancing decision is computed and sent to its children nodes in the tree. The controlling nodes of the downstream domains incorporate the load

[2]The value can be determined by the physical memory available on the node

balancing decisions *before* calculating the load imbalance. However, the actual migration of the objects is delayed until load balancing decisions are calculated globally.

If load imbalance does not occur in a domain, the central node of the domain simply bypasses the load balancing and propagates the load balancing decisions made at its upper levels to all its children.

Load Balance Decision Making

When the central node of a load balancing domain determines that it is advantageous to perform load balancing in its domain, the central node starts to invoke load balance decision making algorithm to calculate a new mapping of parallel objects to compute nodes in its domain.

Depending on the amount of load information available, the central node decides whether load balancing algorithm is invoked in a *fully centralized* or *semi-centralized* load balancing scheme. The fully centralized scheme performs load balancing on the central node with the integrated laod information of its domain. A designated Charm++ centralized load balancing algorithm is invoked on the central node which computes a new map to assign objects to its domain processors. Note that domain processors (other than the central node) represents groups of processors in their subtree. Thus, migration decision is made in the form of groups: obj_i: G_m to G_n, that is object i is to move from group m to group n. The subsequent load balancing subdomains of each group will then decide which *processor* to assign the object. Decision time is reduced because the number of partitions is small (B instead of the total number of processors P, where B is the branching factor of the tree).

Semi-centralized scheme is used when the central node only has limited amount of load information about group members in its load balancing domain due to the memory constraints mentioned. In this case, the central node only make decisions on how much load be transferred from a group to another in its domain. When the group members receive this load balancing decision, they independently select objects to migrate to other groups.

In both schemes, load balancing decisions are only made at group level for each object. In fact, only level 1 intermediate nodes make final load balancing decisions at the processor level.

When sub-domain nodes receives load balancing decisions with message, they incorporate the decision into the load information database accordingly. Since intermediate nodes only carry the load data without the actual migratable objects, no real object migration occurs at this time until the final load balancing decisions are made. Instead, only "migration" of the load data records is performed at those intermediate levels. For example, for migrating object i from group G_m to G_n, the outgoing group G_m sends the object load records of i to G_n, and deletes the records from its load database. The group G_n receives the load records about obj_i from G_m, and adds the records to its load database. After the "migration" finishes, the load database of the subdomain groups is updated for further load balancing at lower levels.

Object Migration

After load balancing decisions are made globally (i.e. from top to bottom in the hierarchical tree), object migration will be performed according to the new object-to-processor map.

At this point, due to the fact that the load balancing decisions are made independently in each load balancing domain and a migration decision may involve other domains, the migration decisions across different domains are incomplete. For example, some processors only know the "from" portions of the migrations, while other processors only know the "to" portions of the migrations. For example, a processor P_i in a load balancing domain G_m knows that an object O that it houses now needs to be migrated to a processor *somewhere* in load balancing domain G_n. Similarly, a processor P_j in G_n knows that it will receive the object O migrated from *some* processor in G_m. Thus, the load balancing decisions need to be integrated so that the complete migration path is formed with specific processors instead of the load balancing domain, so that objects requiring migration can be sent directly to their destination processors. To solve this problem, HybridLB uses a global reduction operation on the hierarchical tree to integrate the load balancing decisions. Any processor with incomplete migration decisions sends its question to its parent, who will try to match the questions it receives from all child nodes and send unmatched questions up to its own parent. When the reduction reaches the root of the tree, all migration decisions should have been matched and the matched decisions are sent down the tree. A distributed hash table can be used for this purpose, although the current implementation does not use that.

Complexity

The load balancing overhead of the hybrid load balancing scheme is distributed across the system, but the distribution is not uniform. The complexity of the balancing process can be analyzed in terms of the algorithm complexity, the communication overhead (number of messages sent), and memory overhead.

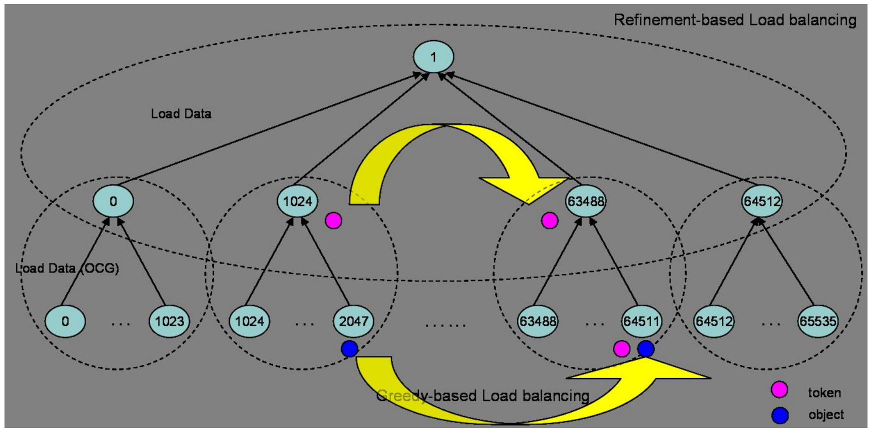


Figure 8.4: HybridLB load balancing scheme

Assuming a peta-scale machine has P processors, and an application has N migratable objects ($N > P$), hybrid load balancing form each load balancing domain with G processors at each level, that is, the tree has a branching factor of G (assuming same factor at each level for simplicity). There are total L levels, where $L = \lceil \log_G P \rceil$.

Computation Complexity:

When load balancing occurs at level i, assuming a greedy-based load balancing algorithm is performed, the complexity is

$$O(N_i(logN_i + logG) + GlogG)$$

where N_i is the number of migratable objects to balance in the given load balancing domain. At level i, $N_i = N/G^i$.

Thus, the total cost of the hybrid load balancing algorithm is:

$$\sum_{i=0}^{L-1} \frac{N}{G^i}(log\frac{N}{G^i} + logG) + GlogG$$

Assuming a binary tree ($G = 2$) hierarchical tree, the complexity of the algorithm is:

$$O(\sum_{i=0}^{L-1} \frac{N}{2^i}(log\frac{N}{2^i}))$$

or

$$O(NlogN)$$

As an example, figure 8.5 shows a plot of CPU overhead for load balancing an application with 1000000 parallel objects on 64K processors with various number of levels of the hierarchical trees.

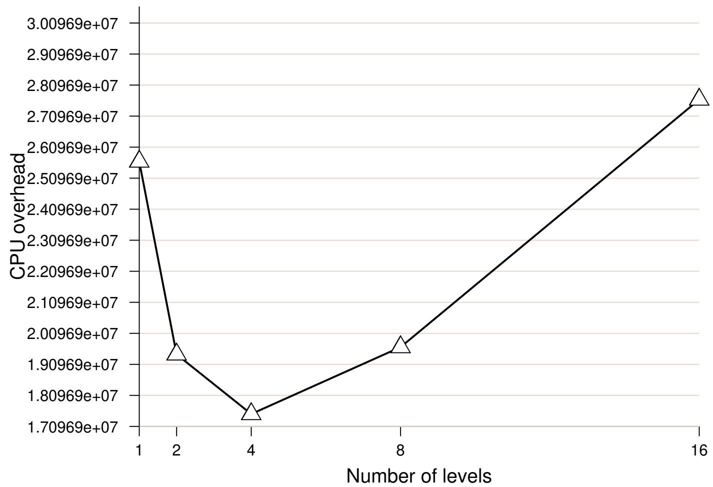


Figure 8.5: Plot of CPU complexity over different levels

Communication Overhead:

When calculating the communication complexity, or the number of messages sent in HybridLB, we consider the messages of the load balancing framework, while ignoring the object migration messages which is subject to both the load balance of the system and the type of load balancing algorithms used.

At a level i of a hierarchical tree, there are P/G^i domains. In the phase of load statistics collection with a full system load update from the leaves to the root, each processor receives G load data messages sent from its children, yielding a total number of

$$\sum_{i=0}^{L-1} \frac{P}{G^i} G$$

, or

$$\frac{PG - G}{G - 1}$$

messages.

In the phase of load balancing decision making from the root to the leaves, there are same number of messages sent:

$$\frac{PG - G}{G - 1}$$

.

Near the end of load balancing step, a reduction is performed to propagate load balancing decisions to all processors which involves the same number of messages. Therefore the total

$$3\sum_{i=0}^{L-1}\frac{P}{G^i}G$$

, or

$$3\frac{PG-G}{G-1}$$

messages.

Assuming a binary tree ($G = 2$) hierarchical tree, the total number of messages is $6P - 6$.

As an exmaple, figure 8.6 shows a plot of communication overhead in terms of the number of messages sent on a 64K processor machine with various number of levels of the hierarchical trees.

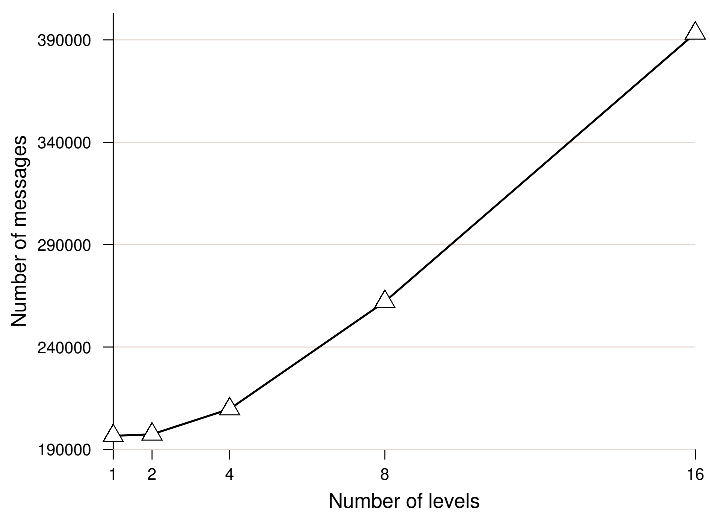


Figure 8.6: Plot of number of messages over different levels

Memory Overhead:

Assuming each object contributes load data of same size (S) in the load database, the total amount of memory needed for storing the database is therefore $N * S$. In HybridLB, load balancing database is distributed in the hierarchical tree. However, the distribution is not uniform as load information gathers as level increases. For example, the root level may have to store the whole database if there is no memory reduction strategy adopted (as described in Section 8.4.1). Assuming database is shrunk at second top level and assuming objects are distributed evenly, the average memory usage to store the database on at that level is

$$NS/G$$

which is also the maximum memory requirement across the whole machine. In general, if database shrinking happens at level i, the peak memory usage on processors at that level is: $NS/(G^i)$, assuming uniform object distribution on processors.

8.5 Related Work

Hierarchical or multi-level load balancing strategies have been studied before in literature. The target applications of the most of these research are applications that assume a continuous flow of subtasks which can be treated as a task scheduling problem or task allocation problem. These applications typically involves a subtask generator with substantial dynamic changing load.

Furuichi et al. [38] present a strategy to partition the search of an OR-parallel graph in a distributed and hierarchical fashion on various processors. Some of these processors function as subtask generators and distribute the tasks among the remaining processors. In this scheme, idle processors request subtasks from those processors. The generation of subtasks is independent of the work requests from idle processors. The subtasks are delivered to processors needing them on demand while achieving load balance.

Ahmad et al. [8] present a semi-distributed strategy for task allocation for regular topologies, e.g., hypercubes as an alternative to completely centralized and completely distributed task allocation strategies. The proposed strategy partitions the compute system into independent regions (spheres) centered at some control points. The central points (schedulers) optimally schedule tasks within their spheres and maintain state information with low overhead. The paper presented an efficient scheme with low messaging overhead for partitioning hypercube systems for task scheduling.

The model of computation in these load balancing strategies is different from ours. These load balancing problem essentially tries to partition the subtask generation or scheduling tasks in a hierarchical way to distribute the load balancing effort. However, due to the dynamic nature of the applications, load balancing strategies easily suffer from the aging of load information as well as the inaccuracy of load estimates. Typically these load balancing methods rely on the applications to provide estimation or hints of the workload of computation.

Our hybrid load balancing problem targets on the applications with relatively persistent work and communication load, which allows a measurement-based scheme to automatic collect work load and communication pattern without users intervention. Load balancing problem essentially is turned into a graph partitioning problem. Section 6.3.4 has discussed numerous sequential graph partitioning algorithms which are suitable for centralized load balancing. There are also a number of parallel algorithms. Among them, multilevel algorithms are widely recognized as the state-of-the-art, as they are able to robustly compute high-quality partitionings quickly. ParMETIS [58] is one of them.

ParMETIS [58] is an MPI-based parallel library that implements a variety of algorithms for partitioning unstructured graphs, meshes, and for computing fill-reducing orderings of sparse matrices. It is an extension of the widely used serial package METIS that includes a number of algorithms for partitioning and repartitioning unstructured domains (graphs). The parallel graph partitioning algorithm used in ParMETIS is based on the serial multilevel k-way partitioning algorithm in [59, 60] and parallelized in [57, 79]. It consists of three phases: graph coarsening, initial partitioning, and uncoarsening/refinement. In the graph coarsening phase, a series of graphs is constructed by collapsing together adjacent vertices of the input graph in order to form a related coarser graph. Computation of the initial partitioning is performed on the coarsest (and hence smallest) of these graphs, and so is very fast. Finally, partition refinement is performed on each level graph which is projected back towards the original graph. METIS and ParMETIS mainly target on partitioning a very large graph into relatively small number of sub-domains, it does not work effectively when the number of target partitions increases to 64,000 for example. The algorithm also tends to be quite expensive, for example, partitioning a relatively small graph with 10K nodes to 1K partitions, Metis takes about 73 seconds on a PC Linux box (Table 7.1).

Other research in hierarchical load balancing method focuses on building a hierarchical tree according to network hierarchy in order to minimize load balancing overhead.

Hierarchical Balancing Method (HBM) [89] is a load balancing strategy that takes advantage of the hierarchical tree for performing load balancing in multi-levels. In HBM strategy, a parallel system is organized into a hierarchy of balancing domains according to network hierarchy, for example a hypercube interconnection as shown in Figure 8.3. Global balancing is achieved by ascending the tree and balancing the load between adjacent domains at each level in the hierarchy. In the example, processors at level l_i receive load information from both its children nodes at level l_{i-1} domains. When load information is received, load balancing is performed at level l_i, and load balancing propagate to higher level of l_{i+1}. Task migration is controlled by intermediate nodes which, upon detecting an imbalance among its children nodes, notify *all* processors belonging to the overloaded subtree to migrate work to the underloaded subtree. Processors within the overloaded branch transfer a designated amount of their load to their "matching" neighbor [3] in the adjacent underloaded subtree. Given a hypercube interconnection, these processors are directly linked to one another, which leads to high efficiency in task migration.

Although the basic idea of the HBM is similar to our algorithm, there are a few significant differences.

1. The task migration in HBM is effective due to the work transfer scheme between "matching processors", it however restricts the load transfer in only one direction from overloaded branch to the underloaded one, which prevents it from balancing the load in a global fashion. For the same reason, the HBM algorithm tends to neglects to optimize the load balancing communication overhead to minimize the distance and volume of work transfered.

2. HBM adopts same load balancing strategy at every level, however, load balancing overhead increases when ascending the tree — at the top level of the tree, the load transfer involves all processors in the system.

3. The load balancing overhead in HBM tends to be high due to the repeated load balancing effort when ascending the tree. Every time when balancing load at a certain level, all the processors in its domain needs to be balanced, although each individual subtree has already been balanced. This behavior could easily result in excessive multiple hops of migration of an object in the system to its final destination, which is not efficient.

Our hybrid load balancer in Charm++ makes use a more flexible tree and allow each level choose different load balancing algorithm to adaptively reduce the increasing load balancing overhead towards higher levels to overcome these drawbacks. The details of hybrid load balancer is presented in Section 8.4.1.

8.6 Performance Evaluation of HybridLB

We evaluate the performance of the new HybridLB using both a real-world execution of LeanMD program and a simulation study with the BigSim simulator we developed.

[3] A processor in the left subtree at a given level has a corresponding processor in the right subtree

8.6.1 LeanMD

We tested the performance of HybridLB with LeanMD on LeMieux with real executions and compared the performance with a centralized and a fully distributed load balancers. The tests were run on 264 processors of LeMieux.

In this test, we ran LeanMD with the BUTANE atom benchmark with 2-away simulation. This BUTANE system is a small atom system having only 256 atoms, 128 angles, 64 torsions and 64 1-4 interactions. LeanMD creates 64 Cell and 2080 CellPair objects for this system.

We compared LeanMD performance with three different load balancers: a greedy-based centralized load balancer (GreedyLB), a neighborhood averaging load balancer (NeighborLB) and a hybrid load balancer (HybridLB). NeighborLB assumes a 2D mesh topology where each processor has 4 neighboring processors. HybridLB uses a 3 level hierarchical tree, with 33 processors forming the level 1 load balancing domains, and 8 processor groups forming the level 2 load balancing domains.

Figure 8.7 shows the result of the comparisons. Each plotted line shows the time per step in seconds for a total of 300 time-steps. The time per step is averaged in 5 timesteps.

It can be seen that the GreedyLB improved the time per step from around 0.02s to only 0.007s, and it performs the best among the three load balancers. HybridLB performed nearly as well as GreedyLB, while NeighborLB could only improve the performance slowly.

Table 8.6 shows the total load balancing time taken by the three load balancers. The NeighborLB is the fastest, as it took only a fraction of time of the other load balancers in comparison. This is due to the distribution of load balancing domains and the small size of each domain (1 center processor and 4 neighboring processors). HybridLB load balancing is about as fast as GreedyLB. The test shows that even for a small machine, hybrid load balancer can still perform almost as well as the centralized load balancer.

GreedyLB	NeighborLB	HybridLB
0.11s	0.005s	0.10s

Table 8.6: Comparison of Load Balancing Time

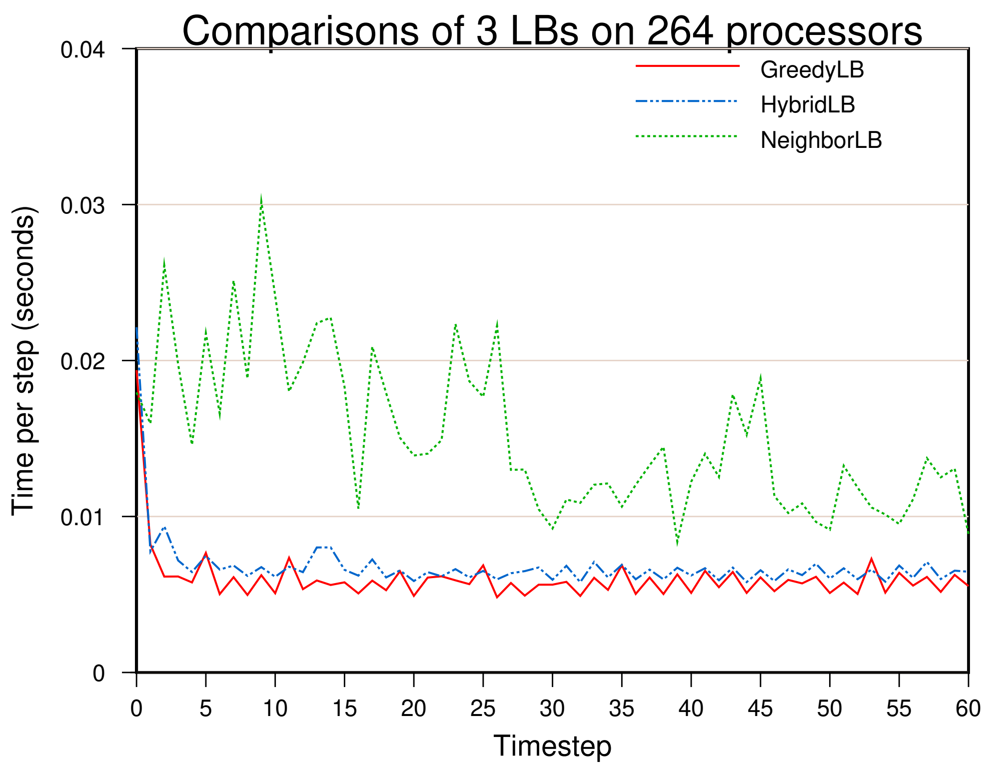


Figure 8.7: Performance comparisons of three load balancers

8.6.2 Performance Study of HybridLB on Very Large Machines via Simulation

We studied the performance of HybridLB running on the BigSim simulator simulating a Blue Gene/L like machine. We ran load balancing benchmark lb_test on the BigSim simulator to simulate the HybridLB load balancing strategy on the target machine with 32K and 64K processors using only 64 processors (16 nodes) of LeMieux to simulate. Lb_test program creates varying number of objects from 128K to 1M. These objects communicate in a mesh 2D virtual topology. The hierarchical tree used by HybridLB is built as following: every 1024 processors form a load balancing domain at level 1, and 32 such domains in the 32K processor simulation form the level 2 load balancing domain; while in 64K processor simulation case, there are 64 such domains. We used greedy-based load balancing algorithms (GreedyLB or GreedyCommLB) at level 1 and use a refinement-based load balancing algorithm at level 2.

Load Balancing Memory Usage

We first measured the maximum usage of memory due to the load database across all the simulated processors for various experimental configurations. Tests are running on the simulator of 32K and 64K processors. In this test, the hierarchical tree used by the HybridLB is two levels. The branching factor at the bottom level is 1024. Figure 8.8 illustrates the hierarchical tree for 32K processors used for these tests. Similarly, a 64K processors tree is shown in Figure 8.9.

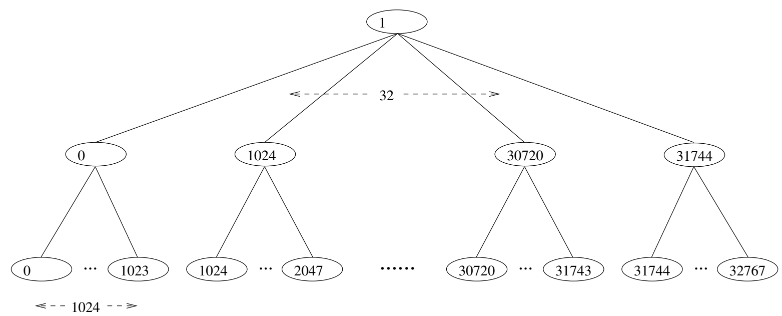


Figure 8.8: A hierarchical tree for 32K processors

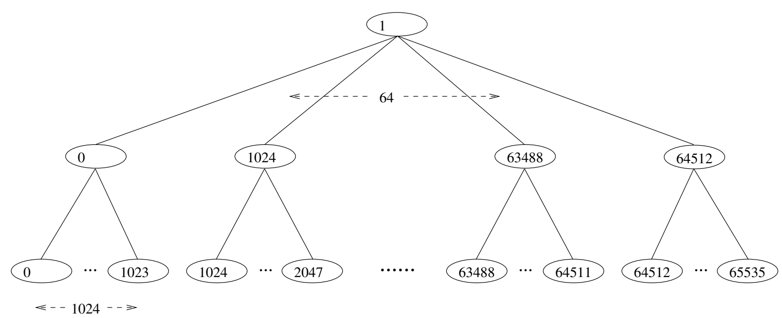


Figure 8.9: A hierarchical tree for 64K processors

The results are shown in Table 8.7. The memory usage reported is the maximum total memory needed for storing the object-communication graph on the central node of a domain. The memory usage of the load balancing algorithm itself is not included.

	Number of Objects		
Number of PEs	256k	512k	1m
32k	3.51M	6.97M	13.87M
64k	1.78M	3.5M	7M

Table 8.7: Maximum memory usage of HybridLB (in bytes)

Compared with the memory usage in the same lb_test with the centralized load balancing strategy in Table 8.1, the memory usage of HybridLB is significantly reduced. Furthermore, in the two cases involving 1 million communicating objects, the maximum memory usage of HybridLB across all simulated processors decreases as the number of simulated processors increases. This is because the number of objects in each load balancing domain is smaller (half) in 64K processor simulation than that in the 32K processor simulation.

Load Balancing Cost in Time

In the next test, we simulated the whole process of a hybrid load balancing strategy on a machine with 64K processors with varying number of migratable objects. The total load balancing execution time is the time taken for the four load balancing steps (Section 8.4.1). The results are shown in Table 8.8.

	Number of Objects		
Strategy	256k	512k	1m
Greedy	1.39	2.79	5.3
GreedyComm	1.39	4.2	5.9

Table 8.8: Total load balancing time on 64K processors (in seconds)

Compared with the results in the same lb_test with the centralized load balancing strategy in Table 8.4, the execution time of HybridLB is significantly reduced. This is largely due to the fact that the load balancing algorithms are running much faster given much smaller load balancing domains and fewer migratable objects to balance. The parallel executions of load balancing algorithms by the independent load balancing domains also improves the performance.

Load Balancing Quality

Table 8.9 shows the maximum predicted load after load balancing for the 64K processor simulation. The maximum predicted load is a performance metric that represents the quality of the load balancing strategy outcome (Section 7.1).

Compared with the results with the centralized load balancing strategy in Table 8.5, the maximum predicted load across all simulated processors after load balancing is only slightly worse (less than 1%). This demonstrates that HybridLB is capable of making load balancing decisions almost as well as the centralized

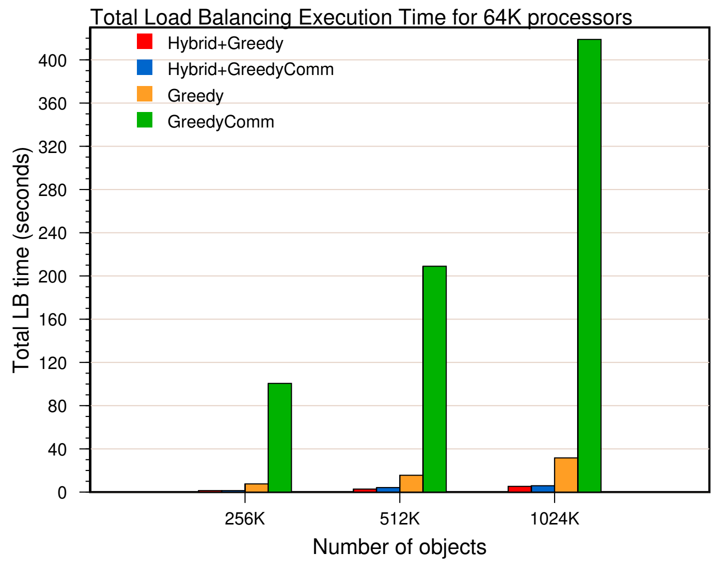


Figure 8.10: Total Load Balancing Execution Time of CentralLB and HybridLB on 64K processors

	Number of Objects		
Strategy	256k	512k	1m
HybridLB with GreedyComm	0.0282	0.0543	0.1077

Table 8.9: Maximum predicted load after load balancing for 64K processors

load balancing strategies, even though it takes much smaller time and memory.

Communication Optimization

We show that our HybridLB strategy optimizes communication better than HBM strategy as shown in Table 8.10. In these tests, the same lb_test benchmark program is simulated for a 64K processor machine with different problem size of 256K and 512K parallel objects. The numbers shown in the table are the total amount of non-local communication volume in byte for each run. HybridLB takes a 3-level tree with branching factor of $64X32X32$ at each intermiediate level. The HybridLB invokes Metis at the lowest level to optimize the communication aggressively.

The result in Table 8.10 shows that the object-to-processor mapping generated by the HybridLB reduced nonlocal message almost in half, which means a much better load balancing decision that takes communication into account.

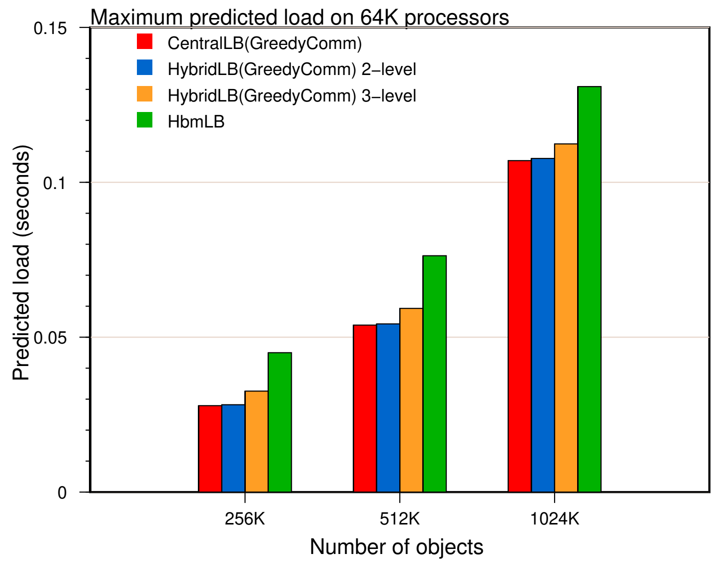


Figure 8.11: Comparison of Maximum predicted load after load balancing on CentralLB HybridLB (2 and 3 level) and HbmLB on 64K processors

	Number of Objects		
Strategy	256k	512k	1m
HbmLB	987.79MB	1.916GB	3.710GB
HybridLB(Metis)	645.46MB	1.113GB	2.107GB

Table 8.10: Non-local communication volume on 64K processors

The Effect of Number of Levels on Load Balancing

It is interesting to see how the number of levels of the hierarchical tree will effect the memory usage, load balancing cost and the quality of the decision. We run the same lb_test benchmark test to simulate a 64K processor machine with HybridLB. In the first test case, HybridLB uses a 2-level tree with a branching factor of $1024X64$, while in the second case, HybridLB uses a 3-level tree with a branching factor of $64X32X32$. The load data is shrunk at the second top level in the tree. We ran the simulation of HybridLB with different problem size of 512K and 1M parallel objects. The results are shown in Table 8.11 and 8.12 respectively.

	memory	decision time	max load
2-level	3.5M	4.2s	0.0543
3-level	6.9M	0.39s	0.0593

Table 8.11: Comparison of 2- and 3-level trees for 512k object case

	memory	decision time	max load
2-level	7M	5.9s	0.1077
3-level	13.9M	1.06s	0.1123

Table 8.12: Comparison of 2- and 3-level trees for 1M object case

We can see from the tables that in both test cases, load balancing in 3-level tree cases uses about two times of the memory as in the 2-level tree cases. This confirms the memory complexity analysis in Section 8.4.1, where the memory usage is $\frac{M}{G}$ for this case, where M is the total amount of memory required to store the global database, and G is the top level branching factor which is 32 for 3-level tree and 64 for 4-level tree.

The decision time in the 3-level case is substantially faster than the 2-level case, however the load balancing quality is about 10% worse. Although it is a tradeoff between decision time and load balancing quality on different levels of trees, for applications of these problem sizes, HybridLB on a simple 2-level tree is sufficiently good. A higher level of tree is not necessary unless the decision time exceeds a threashold, that is the load balancing time becomes greater than the potential benefit of the load balancing.

Chapter 9

Conclusion and Future Work

This thesis aims at developing techniques and methods to facilitate the development of peta-scale applications. We propose employing migratable objects (processor virtualization) for programming peta-scale machines supported by parallel emulation for algorithm validation and parallel simulation for performance prediction, and using new kinds of load balancing strategies to facilitate the development of such applications.

1. We demonstrated that CHARM++ and AMPI with migratable objects are suitable programming models for peta-scale machines in that they are capable of parallelizing applications with sufficient parallelism needed to occupy the large number of processors. The sufficient parallelism is also needed for load balancing to work effectively.

2. We designed and implemented a *scalable* **parallel emulator** of peta-scale machines for evaluating programming models and validating parallel algorithms with large scale of parallelism. The scalability of our parallel emulator is made possible by exploiting a two-level processor virtualization and efficient user level threads.

3. We extended the parallel emulator to a *scalable* **parallel simulator** that is capable of predicting parallel performance to help analysis and tuning of the parallel performance of a given application. We explored parallel discrete event simulation techniques with an optimistic synchronization protocol to improve the accuracy of predicting parallel performance. Our optimization techniques include exploiting the inherent determinacy of an application to reduce simulation overhead.

4. Load balance problem presents significant challenges to applications to achieve scalability on very large machines. We present an enhanced load balancing framework to address the load-balancing

needs of many different types of applications. It provides automatic dynamic load balancing with little user intervention. The framework performs automatic measurement of object computation times and communication patterns including collective communication, without adding undue overhead. We optimized load balancing strategies in multiple dimensions of criteria such as load balancing for improving communication locality, sub-step load balancing, and computation phase-aware load balancing.

5. We studied the limitations of the centralized load balancing strategies for extremely large parallel machines via large scale simulation, and present a new effective hybrid load balancing strategy that combines the advantages of both centralized and fully distributed load balancing strategies. This hybrid load balancing strategy builds load data from instrumenting an application at run-time on both computation and communication pattern in a fully automatic way. It also incorporates a cost function to explicit control the memory overhead of load balancing to make it easy to adapt to extremely large number of processors with even small memory footprint. We study the load balancing performance on comparing trees with varying levels and provide load balancing cost functions to find an optimal tree. We demonstrate that even a small level tree works well in terms of the load balancing cost and the quality of decisions.

Most of our research results are used in real-world applications. Our load balancing framework has been used in many applications such as the well known bio-molecular modeling program NAMD, Car-Parinello *ab initio* molecular dynamics (CPAIMD) simulation, dynamic 3D crack propagation simulation program (Fractography3D), and computational cosmology program (NChilada). NAMD was awarded the Gordon Bell Award at SC02 and later scaled to 1TF of peak performance.

Our research in this thesis also aims for the future. We have built infrastructure including the simulation-based performance modeling tools for performance prediction of very large scale parallel machines, which is useful for exploring new techniques for future machines. As an example, we have explored the dynamic load balancing techniques for very large machines using the infrastructure. One resulting system is a simulation-based performance modeling environment – BigSim – for peta-scale parallel machines as illustrated in Figure 9.1.

The future explorations for this thesis include development of high fidelity simulation model such as utilizing an instruction level simulator to further improve the accuracy of performance prediction.

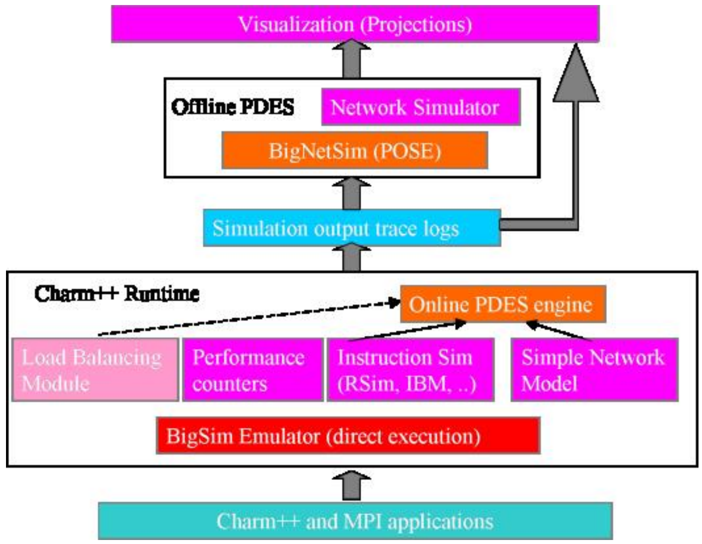


Figure 9.1: The Architecture of BigSim System

Performance analysis tools are very important for understanding and analyzing the performance of parallel applications. The simulator is already able to produce performance data at different levels of detail and interface with performance visualization tools. Projections [54], a performance visualization/analysis tool associated with CHARM++, includes a detailed event log mode and a summary mode that contains concise summarized data. We have employed essentially the same Projections framework for the BigSim simulator. The system can be used for 64K processors, but our experience with runs using Projections exposed a number of bottlenecks and limitations. For example, it is almost impractical to generate 64,000 trace log files for simulating a Blue Gene/L. Reading and writing this large number of log files is expensive in terms of both I/O overhead and memory cost. We plan to explore new methods to enhance the usefulness of the Projections performance analysis tool on large scale parallel machines.

More sophisticated load balancing strategies could be developed under the CHARM++ load balancing framework. For example, load balancing strategies that understand more complicated communication pattern such as all-to-all communication. Integrating the CHARM++ load balancing framework and the communication optimization library [50, 64] could provide load balancing strategies opportunity to optimize the placement of migratable objects for such communication.

As very large parallel machines such as Blue Gene/L are operational, it will be interesting to see how our load balancing strategies perform on these machines.

Fault tolerance is a very important research issue for peta-scale parallel machines. With the extremely large number of processors in peta-scale supercomputers, the probability of one of the processors crashing increases substantially, which potentially affects the scalability of an application on such machines. We have demonstrated in [94] a scalable fault tolerant scheme that uses CHARM++ load balancing capability to maintain the parallel efficiency when fault occurs. It will be interesting to test our fault tolerant scheme on BigSim simulator for very large parallel machines.

References

[1] à la carte - a Los Alamos Computer Architecture Toolkit for Extreme-Scale Architecture Simulation. http://wwwc3.lanl.gov/ parsim.

[2] Center for Simulation of Advanced Rockets at University of Illinois at Urbana-Champaign. http://www.csar.uiuc.edu.

[3] The Advanced Simulation and Computing Program. http://www.lanl.gov/asci.

[4] The GNU Portable Threads. http://www.gnu.org/software/pth.

[5] An Overview of the BlueGene/L Supercomputer. In *Supercomputing 2002 Technical Papers*, Baltimore, Maryland, 2002. The BlueGene/L Team, IBM and Lawrence Livermore National Laboratory.

[6] Vikram S. Adve, Rajive Bagrodia, Ewa Deelman, and Rizos Sakellariou. Compiler-optimized simulation of large-scale applications on high performance architectures. *Journal of Parallel and Distributed Computing*, 62:393–426, 2002.

[7] Gagan Aggarwal, Rajeev Motwani, and An Zhu. The load rebalancing problem. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 258–265, New York, NY, USA, 2003. ACM Press.

[8] Ishfaq Ahmad and Arif Ghafoor. A semi distributed task allocation strategy for large hypercube supercomputers. In *Proceedings of the 1990 conference on Supercomputing*, pages 898–907, New York, NY, 1990. IEEE Computer Society Press.

[9] Ishfaq Ahmad, Arif Ghafoor, and Kishan Mehrotra. Performance prediction of distributed load balancing on multicomputer systems. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 830–839, Albuquerque, NM, 1991. ACM Press.

[10] Rajive Bagrodia, Ewa Deeljman, Steven Docy, and Thomas Phan. Performance prediction of large parallel applications using parallel simulations. *SIGPLAN Not.*, 34(8):151–162, 1999.

[11] Amnon Barak, Avner Braverman, Ilia Gilerman, and Oren Laden. Performance of PVM with the MOSIX Preemptive Process Migration Scheme. In *Proc. 7th Israeli Conf. on Computer Systems and Software Engineering*, pages 38–45. IEEE Computer Society Press, June 1996.

[12] Amnon Barak, Shai Guday, and Richard G. Wheeler. The mosix distributed operating system. In *LNCS 672*. Springer, 1993.

[13] Kevin Barker, Andrey Chernikov, Nikos Chrisochoides, and Keshav Pingali. A Load Balancing Framework for Adaptive and Asynchronous Applications. In *IEEE Transactions on Parallel and Distributed Systems*, volume 15, pages 183–192, 2003.

[14] Kevin J. Barker and Nikos P. Chrisochoides. An Evaluation of a Framework for the Dynamic Load Balancing of Highly Adaptive and Irregular Parallel Applications. In *Proceedings of SC 2003*, Phoenix, AZ, 2003.

[15] A. Basermann, J. Clinckemaillie, T. Coupez, J. Fingberg, H. Digonnet, R. Ducloux, J.-M. Gratien, U. Hartmann, G. Lonsdale, B. Maerten, D. Roose, and C. Walshaw. Dynamic load balancing of finite element applications with the DRAMA Library. In *Applied Math. Modeling*, volume 25, pages 83–98, 2000.

[16] M. Berger and S. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. In *IEEE Trans. Computers*, volume C-36, pages 570–580. IEEE Computer Society, 1987.

[17] Kathryn Berkbigler, Graham Booker, Brian Bush, Kei Davis, and Nicholas Moss. Simulating the Quadrics Interconnection Network. In *High Performance Computing Symposium 2003, Advance Simulation Technologies Conference 2003*, Orlando, Florida, April 2003.

[18] Milind Bhandarkar and L. V. Kalé. A Parallel Framework for Explicit FEM. In M. Valero, V. K. Prasanna, and S. Vajpeyam, editors, *Proceedings of the International Conference on High Performance Computing (HiPC 2000), Lecture Notes in Computer Science*, volume 1970, pages 385–395. Springer Verlag, December 2000.

[19] Shahid H. Bokhari. Dual processor scheduling with dynamic reassignment. *IEEE Trans. on Software Eng.*, 5:341–349, 1979.

[20] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau L. Blni, Albert I. Reuther, Mitchell D. Theys, Bin Yao, Richard F. Freund, Muthucumaru Maheswaran, James P. Robertson, and Debra Hensgen. A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. In *HCW '99: Proceedings of the Eighth Heterogeneous Computing Workshop*, page 15, Washington, DC, USA, 1999. IEEE Computer Society.

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

[22] Robert K. Brunner. Versatile automatic load balancing with migratable objects. TR 00-01, January 2000.

[23] Robert K. Brunner and Laxmikant V. Kalé. Handling application-induced load imbalance using parallel objects. In *Parallel and Distributed Computing for Symbolic and Irregular Applications*, pages 167–181. World Scientific Publishing, 2000.

[24] Robert K. Brunner, James C. Phillips, and Laxmikant V. Kale. Scalable Molecular Dynamics for Large Biomolecular Systems. In *Proceedings of Supercomputing (SC) 2000, Dallas, TX, November 2000. Nominated for Gordon Bell Award.*, November 2000.

[25] Sayantan Chakravorty and L. V. Kale. A fault tolerant protocol for massively parallel machines. In *FTPDS Workshop for IPDPS 2004*. IEEE Press, 2004.

[26] Yaun-Chien Chow and Walter H. Kohler. Models for dynamic load balancing in homogeneous multiple processor systems. In *IEEE Transactions on Computers*, volume c-36, pages 667–679, May 1982.

[27] Mark J. Clement and Michael J. Quinn. Analytical performance prediction on multicomputers. In *Supercomputing*, pages 886–894, 1993.

[28] A. Corradi, L. Leonardi, and F. Zambonelli. Diffusive Load Balancing Policies for Dynamic Applications. In *IEEE Concurrency*, pages 7(1):22–31, 1999.

[29] Scott Pakin Darren J. Kerbyson, Fabrizio Petrini. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Supercomputing 2003*, November 2003.

[30] Samir Ranjan Das, Richard Fujimoto, Kiran S. Panesar, Don Allison, and Maria Hybinette. GTW: a time warp system for shared memory multiprocessors. In *Winter Simulation Conference*, pages 1332–1339, 1994.

[31] K. Devine, B. Hendrickson, E. Boman, M. St. John, and C. Vaughan. Design of Dynamic Load-Balancing Tools for Parallel Applications. In *Proc. Intl. Conf. Supercomputing*, May 2000.

[32] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Bruce A. Hendrickson, James D. Teresco, Jamal Faik, Joseph E. Flaherty, and Luis G. Gervasio. New challenges in dynamic load balancing. *Appl. Numer. Math.*, 52(2–3):133–152, 2005.

[33] Phillip M. Dickens, Philip Heidelberger, and David M. Nicol. A distributed memory lapse: parallel simulation of message-passing programs. *SIGSIM Simul. Dig.*, 24(1):32–38, 1994.

[34] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice And Experience*, 21(8):757–786, August 1991.

[35] D. L. Eager, E. D. Lazowska, and J. Zahorjan. The limited performance benefits of migrating active processes for load sharing. In *Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS)*, Santa Fe, NM, 1988.

[36] Ahmed K. Ezzat, R. Daniel Bergeron, and John L. Pokoski. Task allocation heuristics for distributed computing systems. In *ICDCS*, pages 337–346, 1986.

[37] Richad M. Fujimoto. Parallel discrete event simulation: Will the field survive? *ORSA Journal on Computing*, 5(3):213–230, Summer 1993.

[38] M. Furuichi, K. Taki, and N. Ichiyoshi. A multi-level load balancing scheme for or-parallel exhaustive search programs on the multi-psi. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990.

[39] Anna Ha'c and Xiaowei Jin. Dynamic Load Balancing in Distributed System Using a Decentralized Algorithm. In *Proc. of 7-th Intl. Conf. on Distributed Computing Systems*, April 1987.

[40] Dorit S. Hochbaum and David B. Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *J. ACM*, 34(1):144–162, 1987.

[41] Chao Huang. System support for checkpoint and restart of charm++ and ampi applications. Master's thesis, Dept. of Computer Science, University of Illinois, 2004.

[42] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, pages 306–322, College Station, Texas, October 2003.

[43] Arthur Ieumwananonthachai, Akiko N. Aizawa, Steven R. Schwartz, Benjamin W. Wah, and Jerry C. Yan. Intelligent mapping of communicating processes in distributed computing systems. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 512–521, New York, NY, USA, 1991. ACM Press.

[44] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto. Time warp operating system. In *Proceedings of the 11th ACM Symposium on Operating System Principles*, pages 77–93, 1987.

[45] Rashmi Jyothi, Orion Sky Lawlor, and L. V. Kale. Debugging support for Charm++. In *PADTAD Workshop for IPDPS 2004*, page 294. IEEE Press, 2004.

[46] L. V. Kalé. Comparing the performance of two dynamic load distribution methods. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 8–11, St. Charles, IL, August 1988.

[47] L. V. Kale and Milind Bhandarkar. Structured Dagger: A Coordination Language for Message-Driven Programming. In *Proceedings of Second International Euro-Par Conference*, volume 1123-1124 of *Lecture Notes in Computer Science*, pages 646–653, September 1996.

[48] L. V. Kale, Milind Bhandarkar, and Robert Brunner. Run-time Support for Adaptive Load Balancing. In J. Rolim, editor, *Lecture Notes in Computer Science, Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun - Mexico*, volume 1800, pages 1152–1159, March 2000.

[49] 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.

[50] L. V. Kale and Sameer Kumar. Scaling collective multicast on high performance clusters. Technical Report 03-04, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 2003.

[51] Laxmikant Kalé, Robert Skeel, Milind Bhandarkar, Robert Brunner, Attila Gursoy, Neal Krawetz, James Phillips, Aritomo Shinozaki, Krishnan Varadarajan, and Klaus Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151:283–312, 1999.

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

[53] Laxmikant V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[54] Laxmikant V. Kalé, Sameer Kumar, Gengbin Zheng, and Chee Wai Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop, International Conference on Computational Science(ICCS)*, Melbourne, Australia, June 2003.

[55] A. Kapelnikov, R. R. Muntz, and M. D. Ercegovac. A methodology for performance analysis of parallel computations with looping constructs. *J. Parallel Distrib. Comput.*, 14(2):105–120, 1992.

[56] George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 35, 1996.

[57] George Karypis and Vipin Kumar. A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In *Proc. of the 8th SIAM conference on Parallel Processing for Scientific Computing*, 1997.

[58] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[59] George Karypis and Vipin Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–13, Washington, DC, USA, 1998. IEEE Computer Society.

[60] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96 – 129, 1998.

[61] R.M. Keller, F.C.H. Lin, and J. Tanaka. Rediflow Multiprocessing. *Digest of Papers COMPCON, Spring'84*, pages 410–417, February 1984.

[62] David Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington Department of Computer Science and Engineering, May 1993.

[63] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graph. *Bell System Tech. Journal*, 49:291–307, Feb. 1970.

[64] Sameer Kumar and L. V. Kale. Opportunities and Challenges of Modern Communication Architectures: Case Study with QsNet. Technical Report 03-15, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 2003.

[65] 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.

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

[67] J. K. Lenstra, D. B. Shmoys, and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Math. Program.*, 46(3):259–271, 1990.

[68] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor – A hunter of idle workstations. In *Proc. 8th Intl. Conf. on Distributed Computing Systems*, pages 104–111, San Jose, Calif., June 1988.

[69] J. Liu and D.M. Nicol. *Dartmouth Scalable Simulation Framework User's Manual*. Dept. of Computer Science, Dartmouth College, Hanover, NH, February 2002.

[70] N. Mansour, R. Ponnusamy, A. Choudhary, and G. C. Fox. Graph contraction for physical optimization methods: a quality-cost tradeoff for mapping data on parallel computers. In *ICS '93: Proceedings of the 7th international conference on Supercomputing*, pages 1–10, New York, NY, USA, 1993. ACM Press.

[71] William F. Mitchell. Refinement tree based partitioning for adaptive grids. In *Proc. Seventh SIAM Conf. on Parallel Processing for Scientific Computing*, pages 587–592, 1995.

[72] L. M. Ni and Kai Hwang. Optimal Load Balancing in a Multiple Processor System with Many Job Classes. In *IEEE Trans. on Software Eng.*, volume SE-11, 1985.

[73] Mani Potnuru. Automatic out-of-core exceution support for charm++. Master's thesis, University of Illinois at Urbana-Champaign, 2003.

[74] Sundeep Prakash and Rajive L. Bagrodia. Mpi-sim: Using parallel simulation to evaluate mpi programs. In *Proceedings of IEEE Winter Simulation Conference*, 1998.

[75] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The wisconsin wind tunnel: Virtual prototyping of parallel computers. In *Measurement and Modeling of Computer Systems*, pages 48–60, 1993.

[76] 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.

[77] Hans Sagan. *Space-Filling Curves*. 1994.

[78] Vikram A. Saletore. A distributed and adaptive dynamic load balancing scheme for parallel processing of medium-grain tasks. In *Proceedings of the Fifth Distributed Memory Computing Conference (5th DMCC'90)*, volume II, Architecture Software Tools, and Other General Issues, pages 994–999, Charleston, SC, April 1990. IEEE.

[79] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning (distinguished paper). In *Proc. EuroPar-2000*, 2000.

[80] W. W. Shu and L. V. Kalé. A dynamic load balancing strategy for the Chare Kernel system. In *Proceedings of Supercomputing '89*, pages 389–398, November 1989.

[81] Horst D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2:135–148, 1991.

[82] A. Sinha and L.V. Kalé. A load balancing strategy for prioritized execution of tasks. In *Seventh International Parallel Processing Symposium*, pages 230–237, Newport Beach, CA., April 1993.

[83] Amitabh Sinha and L.V. Kalé. A load balancing strategy for prioritized execution of tasks. In *International Parallel Processing Symposium*, pages 230–237, New Port Beach, CA., April 1993.

[84] Jeff S. Steinman. Interactive Speedes. In *Proceedings of the 24th annual symposium on Simulation*, pages 149–158. IEEE Computer Society Press, 1991.

[85] Jeff S. Steinman. Breathing time warp. In *Proceedings of the 7th workshop on Parallel and Distributed Simulation*, pages 109–118. ACM Press, 1993.

[86] Bernd Stramm and Francine Berman. Communication-sensitive heuristics and algorithms for mapping compilers. In *PPEALS '88: Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, pages 222–243, New York, NY, USA, 1988. ACM Press.

[87] Michael S. Warren and John K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Supercomputing*, pages 12–21, 1993.

[88] Jerrell Watts. A practical approach to dynamic load balancing. Master's thesis, Scalable Concurrent Programming Laboratory, California Institute of Technology, 1995.

[89] Marc H. Willebeek-LeMair and Anthony P. Reeves. Strategies for dynamic load balancing on highly parallel computers. In *IEEE Transactions on Parallel and Distributed Systems*, volume 4, September 1993.

[90] Terry Wilmarth and L. V. Kalé. Pose: Getting over grainsize in parallel discrete event simulation. In *2004 International Conference on Parallel Processing*, pages 12–19, August 2004.

[91] Terry L. Wilmarth. *POSE: Scalable General-purpose Parallel Discrete Event Simulation*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[92] Terry L. Wilmarth, Gengbin Zheng, Eric J. Bohm, Yogesh Mehta, Nilesh Choudhury, Praveen Jagadishprasad, and Laxmikant V. Kale. Performance prediction using simulation of large-scale interconnection networks in pose. In *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation*, pages 109–118, 2005.

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

[94] Gengbin Zheng, Lixia Shi, and Laxmikant V. Kalé. Ftc-charm++: An in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *2004 IEEE International Conference on Cluster Computing*, San Dieago, CA, September 2004.

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

[96] Gengbin Zheng, Terry Wilmarth, Praveen Jagadishprasad, and Laxmikant V. Kalé. Simulation-based performance prediction for large parallel machines. In *International Journal of Parallel Programming*, volume 33, pages 183–207, 2005.

[97] Gengbin Zheng, Terry Wilmarth, Orion Sky Lawlor, Laxmikant V. Kalé, Sarita Adve, David Padua, and Philippe Geubelle. Performance modeling and programming environments for petaflops computers and the blue gene machine. In *NSF Next Generation Systems Program Workshop, 18th International Parallel and Distributed Processing Symposium(IPDPS)*, Santa Fe, New Mexico, April 2004. IEEE Press.

Vita

Gengbin Zheng was born in Beijing, China in 1972. He received his B.S. degree in Computer Science from Peking University, China in 1995.

Continuing at the same university, he completed his M.S. in Computer Science in 1998 with honors. His master thesis was on the design and implementation of an High Performance Fortran (HPF) compiler.

Gengbin started his Ph.D pursuit at University of Illinois at Urbana-Champaign in 1999. During his study, he worked at Parallel Programming Laboratory led by Professor Laxmikant Kale on Charm++ parallel programming language and run-time system. A paper co-authored by him on scaling the molecular dynamics program NAMD was one of the winners of the Gordon Bell Award at SC2002.

On completion of his PhD, Gengbin will join the Center for Simulation of Advanced Rockets at the same university on a post doctoral position.