VERSATILE AUTOMATIC LOAD BALANCING WITH MIGRATABLE OBJECTS

BY

ROBERT KRAEMER BRUNNER

B.S., Case Western Reserve University, 1990

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2000

Urbana, Illinois

VERSATILE AUTOMATIC LOAD BALANCING WITH MIGRATABLE OBJECTS

Robert Kraemer Brunner, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 2000
Laxmikant V. Kalé, Advisor

ABSTRACT

Accessing the power of parallel computation demands efficient load distribution across the processors of the parallel machine. Applications where parallel processing has succeeded most impressively are those where the problem can be easily decomposed into any convenient number of equal-sized partitions. Applications which are irregular or dynamic in their application structure prove more difficult to implement with good parallel efficiency. Usually, a particular application-specific load balancing strategy is implemented for a particular application, using load measures such as data size to detect or control imbalance. We show that treating a program as a collection of communicating objects, measuring the execution time consumed by those objects at run time, and achieving good load balance by automatically moving those objects from processor to processor allow the implementation of efficient load balancers that work with a variety of applications.

Increasingly, parallel computers come in a variety of forms. Non-dedicated parallel computers in the form of clusters of networked desktop workstations are an important source of computing power for many computational scientists. These parallel platforms, however, introduce new challenges for achieving efficient performance. Our techniques address several of the problems faced by applications running on such clusters.

We describe techniques for managing migration of objects in the Charm++ parallel system. Then we describe a framework for creating application-independent load balancers. Finally, we examine several load balancers implemented using this framework and demonstrate their performance characteristics on synthetic as well as real-life applications.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

A major factor slowing deployment of parallel programs is that efficient parallel programs are difficult to write. Parallel programming adds a second dimension to programming: not just when a particular operation will be executed, but where, i.e. what processor will perform it. A vast number of parallelizable applications do not have a regular structure for efficient parallelization. Such applications require load balancing to perform efficiently in parallel. The load in these applications may also change over time, requiring rebalancing. The programmer is left with the choice of either distributing computation haphazardly, producing poorly-performing programs, or spending more development time including load-balancing code in the application.

In recent years, "parallel" computers based on desktop hardware and standard operating systems have become increasingly popular (e.g. [1], [2]). Networks of commodity workstations are making parallel computation available to an expanding group of researchers. Workstation networks present new issues for the application programmer. Now, in addition to application imbalance, a parallel program must be concerned with background load from other simultaneous users. Parallel programs may run on clusters of workstations on an interactive user's desks, where the primary user only permits parallel computation when the computer is not being used interactively. Finally, computational clusters may expand over time, but with the rapid increase in computational power, new processors are likely to be faster than the older machines they are supplementing. To maximize throughput, parallel applications must take all these factors into account.

Work migration is a unified scheme for handling both application-specific and externally-arising load imbalance. The difficulty with existing strategies for migrating work is that either work is repartitioned in an application-specific way, placing the burden on the application programmer, or that automatic migration is supported, but with poor accuracy and/or resolution, due to the lack of application-specific knowledge. Object migration provides a way of performing accurate, fine-grained automatic load balancing. Viewing an application as a collection of communicating objects allows the application to run without concern for what processors execute which objects, or even how many processors are currently in use. This gives load balancing algorithms the freedom to relocate those objects dynamically, so that the load balancer does not have to be designed for the particular application. Like distributed-object architectures such as CORBA and DCOM, our system allows application objects to communicate without explicit knowledge of the current locations of objects. Unlike these architectures, however, our system is designed for finer-grained, tightly coupled collections of objects. The applications we are addressing are made up of objects which know the implementation and interface details of other components, so the complete generality of the component model is not necessary. These applications do require high performance communication (comparable to the communication performance of more traditional parallel communication libraries such as MPI [3]), which is not generally possible with distributed-object technology.

Migration provides the mechanism for adjusting load for diverse applications, but load balancers must also have a mechanism for determining processor loads, and assessing how much load is represented by a particular object. Run-time load measurement provides a general mechanism for determining processor and object loads, so that load balancers can compare object loads without specific knowledge of what application-specific parameters are controlling the load. This method depends on the observation that, just as programs exhibit locality in their memory accesses, objects exhibit temporal persistence of load, so that the future load associated with an object is predicted by its load in the recent past. The Converse [4] run-time system, which supports a number of parallel languages, including Charm++ [5], provides the communication and execution framework which facilitates such load measurement.

The Charm++ object model is particularly well-suited to object-based load balancing. All application code in a Charm++ program is contained in objects. Objects usually have small, well-defined regions of memory on which they operate, reducing the cost of migration compared to process migration or even pre-emptive thread migration systems, where the thread's entire stack must migrate to the new location. Messages are generally delivered to objects, not processors, so the system has the freedom to place those objects on any processors without changing the application code. Using the Charm++ object model, the run-time system measures the work represented by particular objects, rather than deriving execution time from application-specific heuristics. Furthermore, the run-time system records object-to-object communication patterns, so that load balancers can asses the communication impact of migrating particular objects. Migratable objects do impose additional costs for message indirection and forwarding; however, these costs only apply to messages which must travel across processors. An intelligent load balancer can arrange objects to minimze cross-processor messages. Furthermore, Charm++'s message-driven execution model promotes a high degree of overlap between computation and communication, masking the added message overhead, as well as the overhead associated with the Charm++ message-driven scheduling.

1.1 Load Balancing Challenges

Load balancers must address several sources of parallel inefficiency. In this section, we further subdivide the sources of imbalance and discuss how our approach addresses them.

1.1.1 Application Characteristics

Different applications exhibit diverse patterns of load imbalance. Load balancers must be specifically tuned to work most effectively for the nature of the application. A general-purpose load balancer should recognize all of these varieties of imbalance. We divide these load imbalance patterns into three categories: initial imbalance, slow variation, and abrupt variation.

Initial Imbalance

All except the most regular parallel computations require a non-trivial initial balancing. What processor performs what particular part of the complete computation depends on factors such processor architecture, computation to communication ratio, and problem size. Initial load distribution is usually determined in an application specific manner, using some heuristic assumptions about computation and communication costs to compute an initial work distribution. Determining a good heuristic model is a time-consuming task [6], which may have to be repeated for different problem sizes or parallel machines, especially if the parallel machine is heterogeneous or shared. The alternative solution presented in this dissertation is to use the same measurement-based methods used for dynamic load balancing. Objects are initially distributed using some unsophisticated heuristic method. After a short period of normal computation, actual object times and communications patterns are used to determine an accurate load balancing solution.

Slow Variation

Many problems exhibit slow variation in program load. A particular example is molecular dynamics simulation. Simulations typically run for hundreds of hours of real time. During that time, the load distribution may change dramatically, as pieces of the molecule move around the simulation space, changing the number of particles for which each processor is responsible. Therefore, some load balancing is required for efficient execution. However, these changes occur slowly when compared to the execution time of even very elaborate load balancers. Typically the load remains nearly constant for hours at a time, so rebalancing the load every few hours is sufficient to maintain parallel efficiency. Since load balancing is done so infrequently, more time consuming balancer algorithms, possibly requiring considerable communication and several minutes of computation, are practical, since the extra load balancing cost is more than offset by the increased overall efficiency obtained.

Abrupt Variation

More often, irregular applications exhibit sudden variation in load. Adaptive mesh refinement methods are typical examples of this. As the mesh changes, particular regions in the simulation can become much more expensive. Then, as the perturbation moves on, the mesh may become less dense, resulting in less load. Since the computation is still tightly coupled, even one overloaded processor dramatically decreases the parallel efficiency of the program. In such cases, load balancing should be reviewed whenever the mesh changes, and necessary adjustments made to maintain good load balance. Often, the communication in the application can mask a fairly large amount of load balance-related communication, allowing communication-intensive strategies. Some applications, however, may require such frequent load balancing that there is not much time for communication. In these cases, local strategies which only consider the loads on a few neighboring processors, requiring only limited communication, are likely to result in the best overall efficiency.

1.1.2 Background Load

As we will demonstrate in Chapter 3, applications running on non-dedicated parallel machines must consider background load to maximize throughput. One solution to this problem is to detect when the application is sharing the processor with other computationally-intensive processes and migrate work away. Furthermore, if the background process finishes, the application should detect this and move work back to the now-under-loaded processor.

1.1.3 Other Cluster Issues

To effectively deploy parallel programs on non-dedicated workstation clusters, two more areas need to be addressed. First, when we desire to use otherwise-wasted cycles on desktop workstations, the parallel program must sense when a user requires the full use of the machine, and migrate as much load as possible off that machine to avoid interfering with the primary user's activity. Vacating processors under these conditions is similar to the problem of background load, except that the trigger for migrating load is not a drop in throughput, but a user-generated event. Second, the workstations may have different processor architectures or processor speeds. Our method does not specifically address differences in processor architecture, but in our approach, machines with different speeds are easily handled by automatic run-time benchmarking to determine a speed rating for each processor, which gives a conversion factor so that the measured load of objects can be corrected for speed differences.

1.2 A Typical Application

Although the techniques we are developing are applicable to a wide variety of problems, we have mainly been working with scientific applications which form the bulk of computations using traditional parallel computers (such as the Cray T3E, IBM RS6000/SP, and Cray Origin 2000) today. These machines typically provide low-latency communications necessary for achieving good parallel efficiency for the calculations performed by these applications. There is increasing interest, however, in running these applications on clusters of PCs or workstations with high-speed interconnection hardware such as Myrinet, or even commodity hardware such as 100 Mbps switched Ethernet networks. Some characteristics of these applications include the following:

- Computations are iterative, performing similar computations thousands to millions of times.
- Each iteration is relatively short compared to the total execution time, requiring from 1 second to 1 minute on a single processor.
- Iterations cannot be overlapped to any significant extent. Communication dependencies tend to keep the processors approximately synchronized.
- The load does not change rapidly, requiring hundreds of iterations before load imbalance becomes noticeable, except when sudden (but rare) instantaneous load changes occur.
- I/O is not a significant factor in program performance.

Concentrating on applications in use today allows us to focus on the problems limiting their performance, and to obtain immediate feedback on the efficacy of our approach. Furthermore, we can provide support for old applications on new (for those applications) computing platforms, such as non-dedicated workstation clusters. Certainly, other problems, for example, file I/O, are limiting factors in some applications now, but an even larger class of applications have a variety of barriers to achieving optimum performance. By lowering one barrier (load balancing) we improve the performance of many applications today, and contribute to better performance for future applications.

1.3 Related Work

Since load balancing is such a fundamental problem in parallel programming, it has been the subject of a great deal of research. However, most of that research deals with load balancing for particular algorithms or applications. General purpose load balancing was first examined through simulation, and more practically in the context of operating system process migration. More recently, systems providing finer-grained parallelism with varying degrees of automatic load balancing have emerged.

1.3.1 Simulation Studies of Scheduling and Migration

Early studies in efficiently utilizing the processing power of distributed systems dealt with the problem of load sharing. Several researchers have used data gathered by instrumenting operational Unix installations to study the performance of load distribution strategies. Eager *et al* [7] performed simulation studies of several strategies for distributing load. Simulations were run for ranges of transfer (migration) costs typical for Unix systems. Zhou [8] studied the accuracy of using the lengths of various Unix resource queue lengths as an indicator of system load. Later, Zhou proposed several algorithms [9] for sharing load across clusters (including both centralized and neighborhood strategies) and used simulations to study the performance of these algorithms, using typical resource costs determined from trace information obtained from instrumented Unix machines. Leland and Ott [10] also studied the simulated behavior of simple load sharing strategies, using trace information derived from actual Unix systems. Although there are significant differences between the load sharing problem and load balancing of parallel applications (in parallel applications, data dependencies and locality are often more important in determining performance than resource availability), these studies do suggest load balancing algorithms which may also be effective for parallel applications.

Devarakonda and Iyer [11], [12] use instrumented Unix systems to demonstrate that future resource usage can be predicted from past measured performance. Using this approach, Goswami *et al.* [13], [14] describe several load sharing algorithms that base their process placement decisions on the process's resource usage history. Trace-driven simulations show that these methods produce better load sharing than methods which do not use prediction. Baxter and Patel [15] present an algorithm for distributing dynamic parallel task graphs. Task graphs are computed for several parallel applications. These task graphs are statically distributed, and simulated execution traces are generated. These traces are then used to guide a dynamic migration-based load balancing algorithm, which, using only profiling data from previous runs and processor-local information produces simulated performance near that obtained for an ideal algorithm using global information.

The validity of the prediction of the future behavior of Unix processes, and efficacy of prediction-based and profile-based load sharing is strong evidence that a similar approach is valid for object-based parallel programs.

1.3.2 Process Scheduling and Migration

Increasing interest in distributed computing and computational grids [16], have focused new interest on scheduling and migrating processes for heterogeneous collections of geographically separated processors. Slow communications links permit only occasional dynamic migration, so these systems must ascertain detailed information about each processor to find a good load distribution. A number of projects have addressed process scheduling. These efforts have concentrated on effectively sharing a number of processors among several applications, or between applications and interactive users, or to allow process migration for fault tolerance.

Process migration within clusters has been used for load balancing and fault tolerance, for both parallel applications and collections of unrelated processes. These systems typically use load measurement and migration of entire processes to regulate processor loads. The common approach used on homogeneous clusters is to migrate the entire process to another processor. Open files and sockets associated with the process present technical problems for this approach, which have been handled by several of the proposed systems. As the image of a Unix (or any operating system in use today) process is large, it requires significant communication and consequent delay before a process can be migrated.

The V system [17] was one of the first systems to support process management for clusters of commodity workstations. A typical V cluster is composed of workstations, each owned by individual users. Jobs started on a user's workstation are migrated to idle workstations owned by other users, and then migrated away again when the idle machine's owner returns. V was designed to support diskless workstations with a common network file system. Process migration is handled by freezing the process, moving the process's state, and then resuming it at the new location. Load balancing is achieved by periodically migrating processes to equalize the number of ready-processes on each processor, offering no support for finer-grain load balancing within an application. The only support for parallel applications are inter-process communication (IPC) calls.

Sprite [18] was another early effort to support transparent process migration and file access across a network of workstations. In Sprite, each process is initially created on a home processor, which is typically the workstation belonging to the user creating the process. Later, this process may migrate to less-loaded processors, however, the home processor remains a permanent coordination point for some system calls for the lifetime of the process. Sprite reduces migration costs by moving the memory pages of the migrating process only when a page fault occurs on the destination processor. A centralized load balancer migrates processes based on processor load. As with V, the only support for parallel applications in Sprite is Unix-style IPC functions.

MOSIX [19] is a system which provides a Unix-like model to applications, while still providing transparent process migration. It exhibits many similarities to Sprite, in a more Unix-like environment. MOSIX has been adapted to several computing platforms, but the most recent version runs on Pentium-based systems using Linux. As is the case for Sprite, processes in MOSIX are transparently migrated, preserving the memory and file state of the migrating processes. MOSIX supports PVM [20] and MPI [3] parallel programs, as well as sequential applications. MOSIX uses a decentralized load balancer, which uses processor usage statistics to make migration decisions which improve processor load and memory usage. No system support is provided for moving work among existing processes for fine-grained load balancing.

Condor [21] is a process scheduling system aimed at effectively utilizing idle time on non-dedicated workstations. It supports both sequential processes and parallel PVM applications using CARMI [22]. Condor handles process migration through periodic checkpointing. When a processor becomes unavailable, the processes on that processor are halted and restarted elsewhere with the most-recently checkpointed state. Load balancing is addressed by a scheduler, that decides where to place newly-created and resumed processes.

Piranha [23] is another system for utilizing collections of non-dedicated workstations. Piranha provides a master-slave process model for applications. Master tasks, called *feeders*, never leave the processor where they initially run. When a new processor becomes available, a worker task is started on that processor. Communication between feeders and workers is managed by the Linda [24] system. A worker task retrieves work from a shared work pool in Linda tuple-space, and deposits results back into the tuple-space. When a processor becomes unavailable, the Piranha system informs the task, which performs any application-specific cleanup before exiting. Load balancing is managed implicitly by Linda's underlying distribution of the work pool data structure.

1.3.3 Tasks as Objects

Several distributed multi-threaded systems allow objects or threads representing tasks to migrate to different address spaces of a distributed-memory machine. Although not all of these systems are based on traditional object-oriented languages such as C++, the tasks in these systems are related to objects in that they have private state (*e.g.* a thread stack) and code.

UPVM [25] presents a lightweight process model with a PVM-like message-passing library, which supports thread migration independent of an object-oriented framework. PM2 [26] is another migratable-thread system, which treats threads as remote procedure calls, that return some data on completion. These systems simplify the migration of threads by reserving thread stack space in the virtual memory space of every processor, so that pointers to stack items remain valid after migration.

An early object-based parallel programming system is Sloop [27]. Sloop distributed objects are collections of component objects, managed by system-defined container types called *domains*. Applications can use a variety of alignment calls to advise the run-time system on object placement, but the run-time system has the freedom to redistribute objects. Object interactions are monitored so that the run-time system can relocate objects to improve locality. Although a simple load balancing system is mentioned, no detailed description is provided.

CID [28] is a C-based system providing threads and shared data structures in a distributed-memory environment. CID threads are C functions which are passed parameters and may return a value. In addition, the threads have access to program-defined shared data structures. Load balancing is accomplished when a thread is forked. At that time, the run-time system has the option of either executing the function directly, within the currently running thread, or, if another processor is idle, sending the thread to the idle processor for execution. Once the thread starts, it can not be migrated, and once it has been sent to a processor for execution, it can not be redirected again.

Charm [29] is the precursor to Charm++ [5], which is described in more detail in Chapter 2.

Charm programs are composed of chares, parallel objects which are created during program execution, and branch-office chares, objects with a single representative object branch on each processor. Charm implements a form of message-driven execution, where there are no explicit receives. Instead, messages are sent to entry points, C functions which receive the message as a parameter. Newly-created chares exist as chare seeds, messages which contain the intial data for the chare. The load balancer functions by moving chare seeds to less-loaded processors, using one of several load balancing strategies. Once a chare is instantiated from its seed, it can no longer be moved.

Another task-migration system (derived from Charm) is the ELMO [30] system. ELMO adds SPMD-style functions such as blocking receives to the Charm model. ELMO supports checkpointing, compiler-generated packing of simple data structures, user-provided packing of data structures containing pointers, and compiler-generated packing of C function stacks. ELMO objects can be migrated when not executing an object function, or when blocked waiting for a message. Although load balancing is mentioned as one use of the migration facilities of ELMO, the developers have not pursued that aspect, instead looking at the problems of fault-tolerance and debugging.

Cilk [31] is a C-based multithreaded parallel programming system. A Cilk programs is composed of threads, which the run-time system distributes across the available processors. Each thread is passed some input parameters, and a continuation variable where the thread deposits results. Threads are “non-blocking”, which means that threads only suspend while waiting for results from child threads. Since the run-time system does not allow threads to migrate during their execution, the run-time system need not preserve thread state. Load is distributed using a work-stealing scheduling algorithm, in which a processor which has no threads in its ready-queue selects another victim processor and steals a thread from the victim’s ready queue. The selected thread is the ready thread closest to the top-level thread because these threads are more likely to produce more child threads, providing more work for the stealing processor. Cilk’s thread queue has many similarities to Charm’s chare-seed queue, but the way Cilk threads spawn threads and block gives the run-time system additional information which allows estimation of the work associated with a newly-spawned thread. The most recent version of Cilk, Cilk-5 [32] extends the Cilk model to eliminate explicit continuations, allowing the continuation to be inferred by the Cilk compiler. Additionally, Cilk-5 implements a more complex compilation system which allows sequences of threads which run on the same processor to run close to the speed of the sequential code, and several new synchronization primitives which simplify coding. Unlike previous versions, Cilk-5 optimizes its performance by taking advantage of shared memory, so it currently does not run on distributed-memory machines. By making threads non-blocking, moving threads is simplified, but threads cannot have a persistent state which can be accessed by other threads. No other mechanism is provided for sharing data, so any data required by a thread must be passed as a parameter, increasing the amount of data which must be moved to migrate a thread. In Cilk-5, this deficiency is presumably not an issue since the underlying hardware provides (machine-dependent) shared memory support for applications.

Berger and Browne has described a dynamic load balancing system for the CODE graph-oriented parallel programming language [33], [34]. CODE programs are constructed using a graphical user interface to describe a collection of nodes, representing computations, connected by edges representing communication and control flow. Since the CODE program is represented by a graph, it is natural to use a graph-based load balancer to distribute the nodes across the processors of the parallel machine. However, graph-based load balancing is performed statically, without knowledge of the computational costs of the nodes. Therefore, a dynamic work-stealing algorithm similar to Cilk's scheduler is used to alleviate imbalance produced by the static distribution.

Gardens [35] is another system which supports migratable objects. The focus of Gardens is to support adaptive parallelism on pools of workstations, where the availability of workstations varies dynamically. Gardens is based on Active Messages [36], which provides asynchronous communication. There are two kinds of objects in Gardens, *global objects* (GOs), which are objects which encapsulate data, but not work, and are accessible on all processors, and tasks, which perform work, and may block. However, tasks do not communicate directly with other tasks. Instead, they exchange data with other tasks using GOs. Tasks are migrated non-preemptively, by copying the heap and stack associated with the task. The developers describe their load balancing system as "rudimentary" and do not describe any details. Nevertheless, many similarities between Charm++ and Gardens suggest that it might be an excellent target for our load balancing work.

1.3.4 Data as Objects

A number of researchers are developing libraries of C++ objects and templates to support distributed parallel objects. Many of these use the standard facilities of C++ to implement several styles of distributed objects, such as vectors and matrices of arbitrary data types. The application programmer is provided with data distribution directives and mathematical operations to perform common calculations on the data in a data-parallel fashion, similar to the capabilities provided by HPF [37]. Although arrays are a typical feature of these systems, these arrays differ from the object arrays we will describe in that they are arrays of data, not C++ objects. The array elements are typically scalar values, although some of the systems support more general types. Some examples of these systems include Amelia [38], which supports user-defined types, POOMA [39], which provides a variety of data collection types, such as lists of particles and banded arrays, and DOME [40].

DOME is unique among these systems for providing dynamic load balancing. Applications may specify that certain data arrays are "dynamic". These arrays are initially distributed using a block decomposition. During program execution the operations on these objects are timed and the data they contain is re-mapped to balance the load. Load is rebalanced regularly by counting the number of DOME operations executed by each processor, and rebalancing after the completion of a certain number of operations. However, since the load balancer only has control of array data, only those applications whose computation is dominated by DOME operations can be load balanced with this mechanism.

1.3.5 Automatic Parallelizing Compilers

Parallelizing compilers must also deal with load balancing, although the data-parallelism most commonly exploited by parallelizing compilers is usually more regular and consequently easier to balance. One example of such a compiler is described by Lucco [41], which compiles a subset of Fortran D. Unlike most compilers, which use compile-time information only to distribute load, this compiler also uses run-time information to dynamically repartition data arrays among processors.

On symmetric multi-processors (SMPs) parallelizing compilers often target kernel-level threads packages such as Pthreads [42]. IML [43] is an example of a recently developed thread package for Intel-based SMPs running Windows NT. IML is specifically designed to be a target for parallelizing compilers. IML provides a system of user-level threads, which run inside kernel threads. It provides some simple dynamic load balancing capabilities by allowing underutilized kernel threads to obtain work (user-level threads) from work queues in other threads. Future versions of IML will also include support for migratable threads.

The Dynamic Resource Management System (DRMS) [44] is a system for managing tasks generated by various parallel programming paradigms, including SPMD-style programs in Fortran, C, and C++, and HPF compilers. Rather than just moving work between existing tasks, DRMS directs applications to reconfigure the number of active tasks in use, expanding or contracting the number of tasks to accomodate the available resources. DRMS defines an API which allows programs to declare when they are at *schedulable and observable points* (SOPs). A task which is at an SOP must be able to migrate its state or divide or merge its state in order to change the number of active tasks. How language run-times handle these requests is defined outside of the DRMS model, giving DRMS the potential to operate with many different parallel languages. Three simple schedulers are described, which operate by considering the overall processor loads across the system. These schedulers do not take characteristics of individual tasks, in particular, their patterns of communication, suggesting that this system is better suited to coarse-grained applications, or for overseeing applications or parallel languages which optimize communications using other means.

1.3.6 Application Load Balancing

The vast majority of work in load balancing deals with application-specific load balancing. Application-specific load balancing work ranges from load balancing for a particular, widely-used program, or a particular algorithm, independent of an application, to load balancing techniques that operate on a variety of algorithms that can be cast into a common form (such as graph partitioning) to load balancing for common operations that are an expensive part of many algorithms (such as matrix operations). Application-specific load balancers often achieve excellent results; however, the techniques used are rarely transferable to a wide variety of other applications.

Since parallel applications can be cast as macro-dataflow graphs, graph-partitioning algorithms have a more general value that extends beyond applications where they naturally arise. A variety of applications in areas such as VLSI layout, finite elements, and computational fluid dynamics have used graph partitioning for distributing load. We have implemented methods similar to that described by Farhat [45], and made use of the freely-available METIS graph partitioning library [46] in our load balancing strategies.

1.4 Contributions of this Dissertation

This thesis introduces, develops and elaborates an automatic load balancing method with load measurement and migratable objects. Our research shows that this method works for a variety of applications. The main contributions of this dissertation include the following:

- Object arrays in Charm++, which simplify the creation of migratable objects applications. While relieving the application programmer of the burden of writing load balancing code for the application, writing migratable object code does impose on the programmer some additional work not required in a sequential program. Several synchronization, coordination, and information-sharing mechanisms we developed simplify programming with migratable objects.

- A load balancing framework that performs automatic measurement of object times and communication patterns, as well as tracking total processor usage and idle times, without adding undue overhead. This framework also includes a machine-monitoring interface which can sense changes in background machine load and interactive machine use, allowing load balancing strategies to handle load imbalance due to external effects.

- A demonstration of the value of these techniques by applying them to a to a production parallel molecular dynamics program, resulting in the highest reported speedups (855 on 1536 processors) for MD simulations of biological molecules.

- Implementation and benchmarks of several load balancers which address both application imbalance and externally-generated imbalance.

1.5 Dissertation Outline

Chapter 2 introduces the Charm++ parallel programming language, and features which make it well suited for automatic load balancing. Then we discuss the design and implementation of the Charm++ object array construct that serves as a foundation to our approach. In Chapter 3 we examine load balancing problems in more detail, including some examples. Chapter 4 discusses the load balancing framework we have designed for implementing general-purpose load balancing strategies. Next, in Chapter 5 we present a case study of our molecular dynamics program, NAMD, which uses object measurement and object migration for load balancing. In Chapter 6 we present several load balancing strategies and examine their performance on applications. Finally, in Chapter 7, we look at future directions for this research.

CHAPTER 2

CHARM++ AND MESSAGE-DRIVEN EXECUTION

Object-based migration offers a number of advantages over other migration methods. Unlike with thread or process migration, individual objects have a smaller amount of memory requiring migration. Although the general case is to move the entire memory space of the object, the amount of data migrated is reduced further when the application provides object methods which allow the programmer to only pack the object data that is currently in use, and discard data which is no longer in use.

Object-based migration also provides better granularity of tasks than process migration, allowing fine control over load balancing. There are only two possible loads for a processor in a process migration system, busy or idle. Typically, an object-based application will have many more objects than processors, so by moving some of a processor's objects elsewhere, the processor load is adjustable in many fine-grained steps. Object-based migration also allows better CPU utilization during migration. As soon as a single object has arrived, it can overlap useful work with the transit time for the rest of the objects being migrated.

2.1 The Charm++ Object Model

Data-driven execution [47], [48], [49] is a parallel programming paradigm in which the order of computation is determined by the presence or absence of the data required to perform those computations. In contrast, the order of computation in a typical SPMD-style message-passing application is determined by the order of receives in the program. A message-passing program is

```
T1();
WaitFor(M2);
T2(M2);
WaitFor(M3);
T3(M3);
```

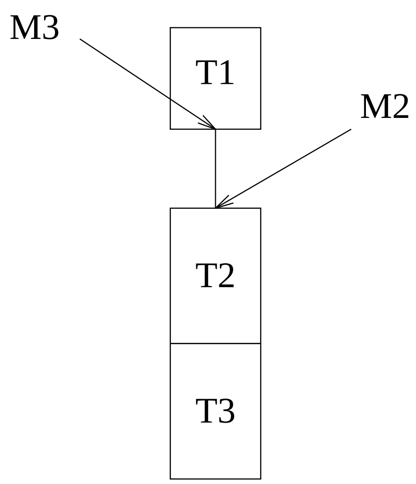

Figure 2.1 An SPMD program which generates unnecessary idle time due to the order of code execution.

composed of computations followed by data exchange followed by more computations. The order of sends and receives often introduces unnecessary data dependencies to the program. Figure 2.1 demonstrates this problem. Function `T1()` must be called, followed by `T2()` and `T3()`. There are no actual data dependencies between any of the functions, but due to the order of program instructions, idle time is introduced. Of course, the program can avoid the added idle time, as demonstrated in Figure 2.2, but only with more complex code.

Data-driven execution avoids the false dependency problem by executing subtasks in a program as the data required by those subtasks becomes available, without adding code complexity like that of Figure 2.2. Charm++ is a C++ based parallel programming language for implementing programs to take advantage of the data-driven execution paradigm. It provides a means of expressing the structure of the macro-dataflow diagram of an application, without introducing artificial dependencies. Figure 2.3 shows a Charm++ object definition for executing the code fragment of Figure 2.1. Charm++ programs are composed of C++ objects containing normal methods and remotely-invokable *entry methods*. Message sending calls in Charm++ are non-blocking, and there are no explicit receive calls. Messages are sent to a particular entry method of a particular object. In Figure 2.3, the object constructor executes `T1()`. After the constructor has finished, a system scheduler selects another message. When `m3` arrives it is placed in the scheduler's queue, and the system subsequently calls `Call_T3()`. When `m3` arrives, it is also enqueued and `Call_T3()` is exe-

```
T1();
done = false;
while (!done) {
  if (Probe(M2)) {
    T2(M2);
    WaitFor(M3);
    T3(M3);
    done = true;
  else if (Probe(M3)) {
    T3(M3);
    WaitFor(M2);
    T2(M2);
    done = true;
  }
}
```

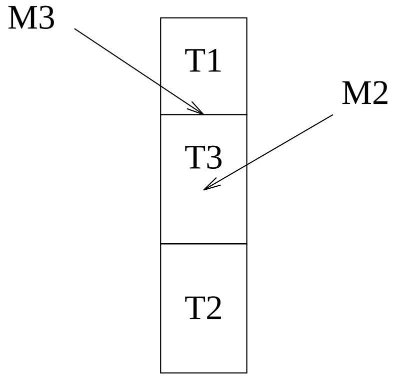

Figure 2.2 An SPMD program which corrects the problem demonstrated in Figure 2.1, at the cost of added code complexity.

cuted. The idle time of Figure 2.1 is eliminated, without the complexity of Figure 2.2. More details on the advantages of message-driven execution can be found in [48].

Charm++ programs are composed entirely of objects, which are instances of C++ classes in which entry methods are invoked by the receipt of messages directed at that entry method. Sending a message to an entry method may be viewed as invoking a method on a remote object. Charm++ assumes a distributed-memory model, although a particular application may use shared memory when provided by the machine with potential loss of portability. When a method is invoked, it runs until completion. Then the next message is retrieved from a scheduler queue and the indicated

```
class MyClass: public Chare {
public:
  MyClass() { T1(); };

  void Call_T2(M2* m2) { T2(m2); };

  void Call_T3(M3* m3) { T3(m3); };
};
```

Figure 2.3 A Charm++ object definition for implementing the code fragment in Figure 2.1.

method of a particular object executed.

There are three types of parallel objects in Charm++. *Chares* are single instances of objects. Chare creation results in the creation of a *seed message*, which is enqueued in the message queue on some arbitrary processor. The seed message may move among the processor message queues to provide the lowest-level of load balancing. Eventually the seed message is dequeued, the chare is created and its constructor called. Once the chare is created, it may send its chare ID to other objects, who may then use that ID to invoke methods on that chare. The second type of Charm++ objects are *object groups* (also called *branch-office chares*). Creation of an object group results in the creation of one object instance on each processor, identified by a *group ID*. Individual objects in an object group are accessed using the pair of group ID and processor number. Other objects may invoke entry methods on any particular processor of an object group, or may broadcast method invocations to every member of the group. Furthermore, any object may use the group ID to obtain a pointer to the branch object on that processor, which may then be used to invoke non-entry methods of the object, or access public data members of the object. The third type of objects, *object arrays*, are arbitrarily-sized sets of objects, where the size is defined when the array is created. Array elements are accessed by remote method invocation, using their *array ID* and element index. Object arrays are described more thoroughly in Section 2.2.

Figure 2.4 shows various objects in a Charm++ program. Several chares have been created on the two processors, and several chare seeds are awaiting creation in the message queue. Two object groups, `L` and `M` have been created, each with one object on each processor. Finally, two object arrays, `Q` and `R`, have elements distributed across the processors.

Decomposing applications into Charm++ objects produce several advantages for automatic migration and load balancing. First, messages to chares and array elements refer to particular objects, not to processors. Therefore, the run-time system has the freedom to relocate those objects without informing any object that may have an object ID. Also, object methods are executed non-preemptively. This dramatically simplifies object migration. Since migration can only occur between method invocations, no stack data needs to be moved to re-create the state of the object. Object state is limited to the data members of the object, resulting in more efficient migration. Because of the way object groups are accessed, they are more difficult to migrate transparently.

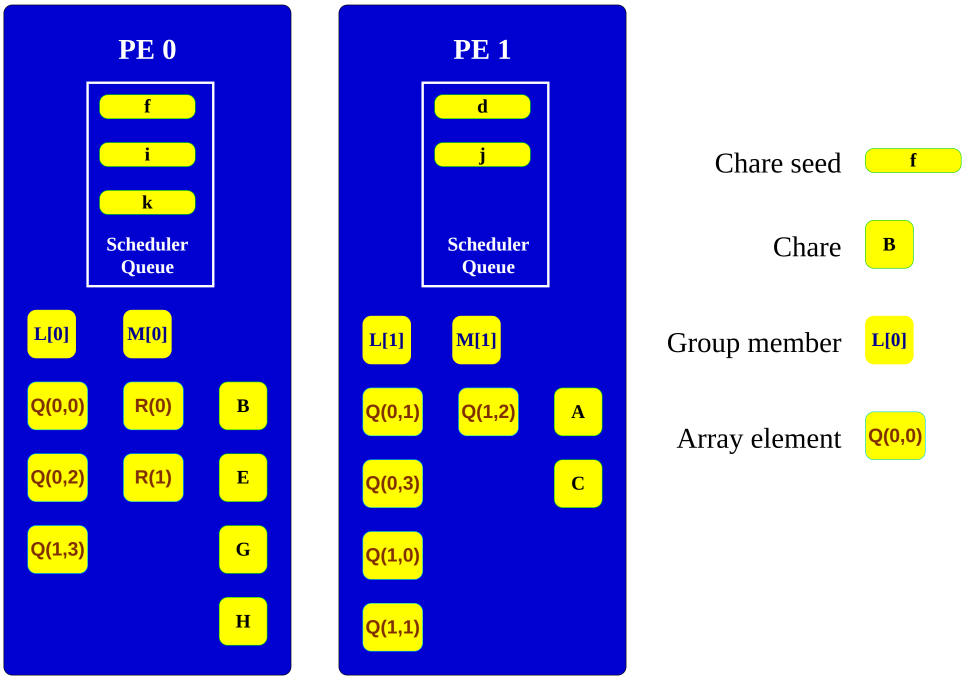

Figure 2.4 A typical object distribution on two processors in a Charm++ program.

An analysis of how object groups are used in applications shows that it is not necessary to migrate object group members. Object groups are employed for one of two purposes. First, object groups may be used to break a problem into parts for parallelization. In this case, replacing the object group with an object array of a desired size and dimension is more convenient for application design as well as migration. The second use of object groups is as a local coordination object for collective operations. For example, a reduction library would be implemented as an object group. Client objects using the reduction library make a function call to the local representative of the object group. If the client object migrates, it may easily obtain the local reduction representative in its new location. For this use, it is important that there is exactly one branch of the object group on each processor.

The way objects are triggered by messages allows the Charm++ run-time to know which object is executing, and for how long it executes. The run-time system can also record which objects send messages, to which objects the messages are delivered, and the message size. This information can be logged and used to guide object migration without knowledge of what computations the objects actually perform.

2.2 Object Arrays

Object arrays are constructs for increasing parallel programming efficiency, by simplifying the process of distributing work among available processors. A program utilizing object arrays receives the benefits of message-driven execution, independence from the number of available processors, and, with assistance from the run-time system, easy migration of work to achieve better load balance.

An earlier implementation of object arrays in Charm++ was described by Krishnan and Kalé [50]. The primary distinction between that implementation and the current one is that the array elements in the prior implementation were designed to move under application control, to improve data locality during various stages of an algorithm. The new design was specifically undertaken to allow elements to move with minimal application interaction, so that efficiency issues can be automatically handled by the run-time system.

2.2.1 Arrays as a Mechanisms for Parallel Program Design

Object arrays simplify the creation of parallel programs which run most efficiently on a wide variety of parallel platforms. Even without object migration, several factor contribute to improved performance. First, object arrays are arrays of message-driven objects. A wide range of applications derive significant benefit from message-driven execution. Second, object arrays promote greater problem granularity, which can provide performance benefits. Many MPI-style codes typically divide problems into P partitions on a P-processor computer. When the problem does not divide into P equal-size partitions, poor load balance results. Also, such codes must explicitly pipeline communication and computation, additional code complexity which is often avoided at the cost of parallel efficiency. An object-array program divides a problem into a number of pieces not tied to the number of processors. Encouraging the user to parallelize his algorithm into the smallest-sized pieces which amortize the communication cost, rather than just distributing one chunk to each processor results in programs which perform well on diverse parallel architectures.

Object arrays allow the application programmer to select an object topology which best represents the particular problem. The most obvious benefit of this is that it provides the application programmer with a convenient way of referring to array elements. For example, an application using a block decomposition of 2-D data could use a 2-D object array. Then the common operation of sending data to a neighbor could use address the message to `(my_i + 1, my_j + 1)`, rather than `((my_i + 1) * row_size + my_j + 1)`. The other benefit is that the load balancer can use the topology information to infer general communication patterns. A load balancer may more easily observe that every element of a 2-D array is mainly sending messages to its four neighbors by knowing the dimensions of the array. That information may cause the load balancer to select a block distribution of objects to minimize communication.

2.2.2 Run-time Support for Object Arrays

The run-time support for object arrays consist of three components, as described in Figure 2.5. The main object is the *array manager*. This object is a Charm++ object group, so there is exactly one branch of the object on each processor. It is responsible for creating the other components of the array, forwarding messages, and supervising the packing and sending of migrating elements. It also handles interactions with the load balancing framework, to be described in Chapter 4. The application programmer selects one of several library-supplied manager types, according to how many dimensions the array requires. The second component is the map object. The map object is also an object group, which provides an initial mapping of objects to processors. The third component is the user-defined *array element* class. The manager creates one array element instance for each element of the array. The array element performs computation and communication with other objects, and the programmer may provide object-specific pack/unpack methods for cases where simply copying the object's data is not sufficient to reproduce the system state.

Figure 2.5 also shows how a message is delivered to an element when no migration has occurred. When an object wants to send a message to an element, it makes a local function call to the local branch of the array manager for the destination element. If no migration has yet occurred, the local manager branch knows on which processor the destination element currently resides. The sending manager branch sends the message to the destination manager branch, which verifies that the element is still present on the processor and delivers the message.

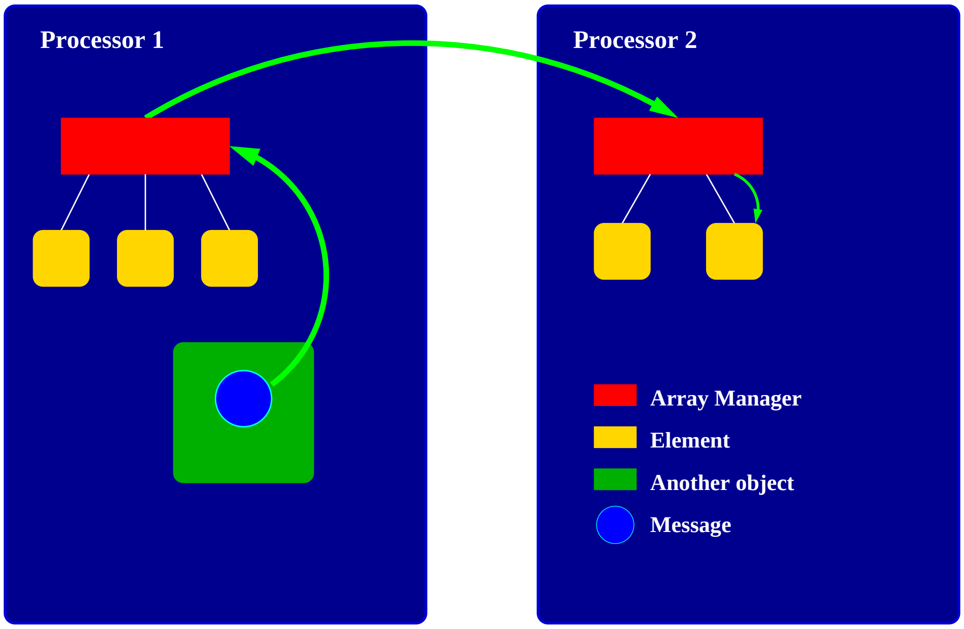

Figure 2.5 Objects created for a single array of several elements, showing a message being delivered for a particular element.

Message Forwarding

When elements migrate, it becomes possible for messages for an object to be delivered to the wrong processor. The most logical solution to insure delivery of such messages is for the run-time system to provide message forwarding, so that incorrectly delivered messages get automatically directed to the processor where the object currently resides. Alternatively, we might have chosen to implement migration as a global synchronization. Such an algorithm would force all messages to be delivered before an element could migrate. Then one or more elements would be migrated, and global communication would be used to inform each processor of the new location of the migrated objects. Finally, normal operation would resume, with every sending processor knowing the updated locations of the migrated objects. Unless migration is always extremely rare, the time required for synchronization with this method is prohibitive.

The main requirement for a forwarding algorithm is for the common case, delivery to an element which has not recently migrated, to occur with only a single message. This requires that each processor have a object location map, containing the (recent) location of any receiving object. When an object moves, every processor's table ideally should be updated (which would be equivalent to the above algorithm), but efficiency demands less frequent updating. There are several methods for implementing message delivery with a stale object map. The following methods all assume that messages contain a *hop-count* field, which can record how many times the message was sent from one processor to another.

One method of updating the object map is to have every migration result in broadcast to all processors, informing them of the migration. The object destination could be broadcast as it arrives at its new processor. Since every processor receives migration updates, most messages can be delivered in one hop, unless a migration has occurred recently and the update has not yet arrived. If a message is delivered to a processor which the destination object has just left, that processor forwards the messages to the object's new home. The disadvantage of this method is its communication cost; every processor receives notification of every move, even though some processors may never need to send a message to some objects.

Another message forwarding algorithm is illustrated in Figure 2.6. In this figure, an object migrates to a new processor, and messages are forwarded as follows:

1. *Obj* migrates from processor B to processor C.

2. In this algorithm, each object has a *home processor*, selected by applying a hash function to the object's index. The home processor is always informed whenever the object migrates. In this case, C informs *Obj's* home processor, D, that the object has migrated.

3. Processor A sends message M to *A's* last known position of *Obj*, B.

4. B no longer has *Obj*, so it forwards the message to *Obj's* home processor, D.

5. D, having been informed of *Obj's* migration in step 2, forwards the message to C.

6. C informs A that it currently has *Obj*, so that A can send future messages directly to C.

If the element has recently migrated, this algorithm requires three hops to deliver the message. However, after sending a message to *Obj* requiring more than one hop, C updates A, so that future messages will be delivered directly, until *Obj* migrates again.

Of course, in this example, B knows that it sent *Obj* to C, so B could forward the message directly to C, suggesting the forwarding algorithm we chose to implement. The top of Figure 2.7

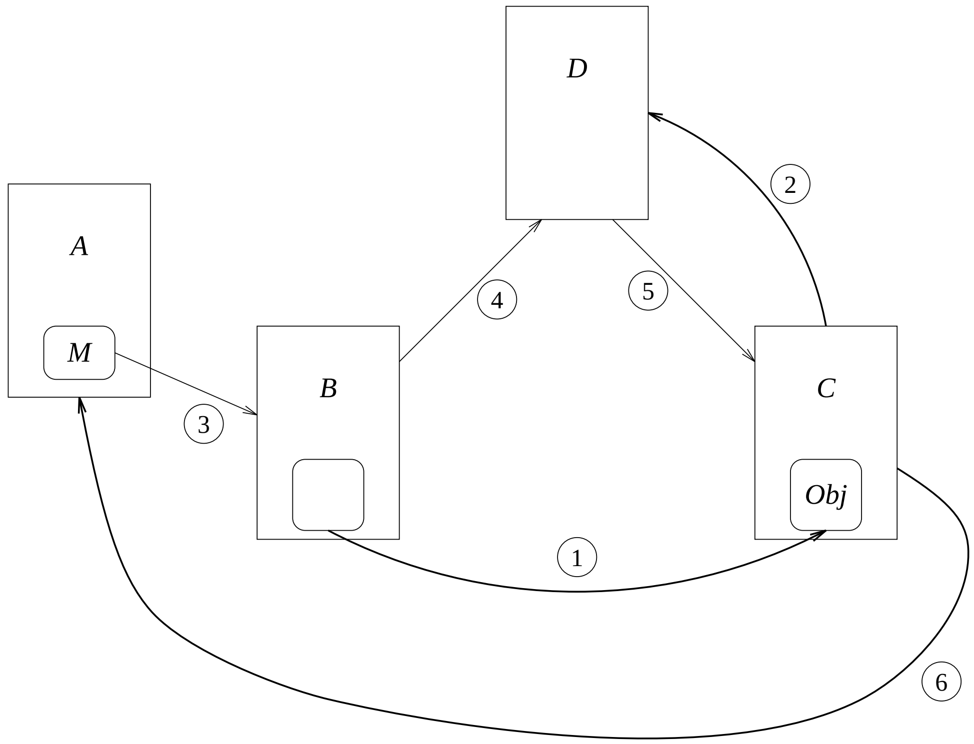

Figure 2.6 One possible messages forwarding algorithm, showing an object moving from processor B to processor C, followed by processor A sending message M to the object.

shows the shortcut procedure described above. The bottom half of Figure 2.7 shows what happens when an object has migrated twice since A last sent to Obj.

1. Obj migrates from B to C.

2. Obj migrates from C to D.

3. A sends a message M to Obj, which, as far as A knows, is assigned to B.

4. B sent Obj to C, so it assumes the Obj is still on C, and forwards M there.

5. C forwards M to D.

6. Since M required more than one hop to be delivered, D sends an update to the originator of the message, A.

Every processor knows either the original processor of each object, or a processor to which that object subsequently migrated, allowing forwarded messages to follow the migrations (*migration chain*) until they catch up with the object. This forwarding algorithm is similar to the method used by the TARMAC mobile memory system [51].

As was the case in the previous algorithms, the most common case of an object that has not migrated recently requires a single send. For the next most common case, where the object has migrated only once, this algorithm requires only two sends, whereas the home-processor algorithm required three. For cases where an element has migrated N times recently, however, this algorithm requires $N + 1$ sends, whereas the home-processor algorithm still requires only three. However, that case is extremely unlikely to occur often in real applications.

Several variants of this algorithm can be pursued if applications reveal weaknesses in the current implementation. For messages requiring a large number of hops, the updating procedure (step 6) could be modified to inform processors in the migration chain, in addition to the originating processor. By updating processors near the start of the chain, future messages (originating at other processors) would be redirected to the destination processor with fewer hops. Also, hybrids of the three methods described here may prove useful. The first method (informing all processors when an object migrates) could be modified to inform some subset of the processors, perhaps based on the recent history of which processors had sent messages to that object. This would optimize

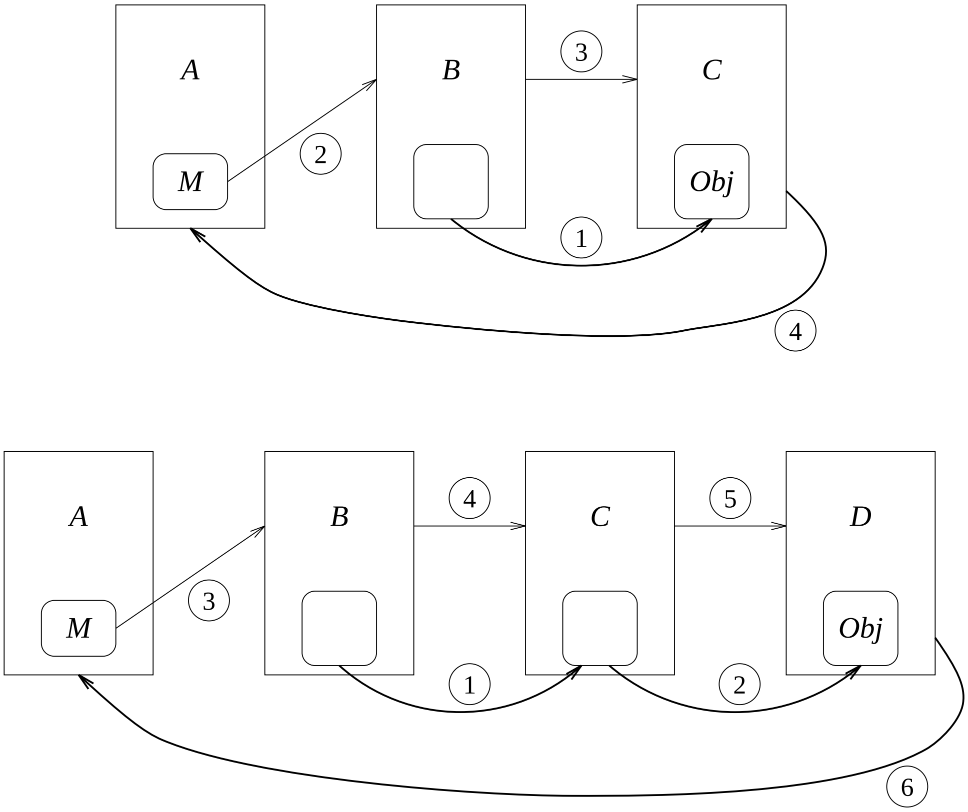

Figure 2.7 Two examples illustrating our algorithm. The top half shows a message M being sent from processor A to object Obj, which has migrated from B to C. The bottom half shows A sending M to Obj, when Obj has migrated from B to C, and then from C to D.

some forwarding traffic by providing direct delivery for more messages, while still using one of the other two algorithms as the fall-back method. More promising is adding some features of the home-processor algorithm to the algorithm we implemented. Messages would follow the chain of migrated elements for some number of hops; then the message would be delivered to the home processor which delivers it to the destination object's processor directly.

None of these algorithms can guarantee to deliver a message, should an object keep moving before the forwarded message catches up with it. Fortunately, this is an idiosyncratic case, since such an object would be migrating so frequently that it would have no time to perform any computation. Also, in parallel computers where message order is not guaranteed, messages would eventually catch up with their objects, due to stochastic factors in message delivery. Nevertheless, this condition can be avoided by temporarily suspending all migration until messages are delivered, which could be triggered whenever any message hop count exceeded some threshold. So far, however, no applications have demonstrated this behavior. Another problem arises when message order is not guaranteed, in which a message may pass up the migrating object in transit. In this case, when the message arrives at a processor, the processor determines that it does not have that object and forwards it on, either to the object's home processor (for the home-processor algorithm), or to a processor earlier in the migration chain chain (for the migration-chain algorithm). Eventually, the object arrives, and then later the message returns for delivery to the object.

Reductions

By providing efficient reduction and multicast operations across the elements of an array, the run-time system can simplify the creation of application programs. In the presence of migrating elements, however, efficient implementations of these operations is complex.

Consider possible implementations of a reduction operation for arrays. One simple implementation would be to have each object send its data to a single supervisor object. This is obviously a poor solution, because it makes the supervisor a bottleneck. A slightly more complex solution is to make each element send its objects up a span-tree which spans the other elements. This solution avoids the bottleneck problem, but results in an increased number of off-processor messages. For example, two elements currently assigned to the same processor may each have different parents on the same remote processor, requiring two messages where one would suffice.

In the absence of migration, an efficient solution for reduction is to collect the reduction data for all the elements on each processor, combine them, and then complete the reduction using a span-tree across processors. However, additional complexity arises if a reduction occurs while an element is migrating. Suppose a migrating element arrives while the local reduction stage is occurring. If that new element has not yet deposited reduction data, the local reduction needs to wait until the element's data is ready. If the element already deposited its data at the previous processor, it would be an error to include the data a second time. Furthermore, an element may arrive which has yet to deposit its data, but the local reduction was already complete, so the data has already been sent up the span-tree.

The method we have implemented is a modification of the efficient scheme described above. The array manager serves as the reduction controller. Each reduction is assigned a serial number (it is assumed that all elements will participate in the same reductions in the same order). Most of the time, the local array manager waits for data from the local elements, and from some off-processor reduction controllers (children in the reduction span-tree). When all of the data has been collected the results are sent to the span-tree parent reduction controller. When a new element arrives, the serial number of the last reduction it participated in is included in the migration data. If the local reduction is still in progress, and if the serial number indicates that it has not yet participated in the current reduction, the reduction controller notes that it must wait for data from this element. If the local reduction has already finished, when the new element submits data, it is sent in a separate message to the reduction controller at the root of the span-tree. Along with each reduction message, the controller adds a counter indicating how many elements have submitted data so far. The root controller then determines that if it receives all the span-tree messages, but the element count does not match the total number of elements of the tree, it must wait for follow-up messages from migrating elements.

Multicasts

Multicasts (including broadcasts) have efficiency issues similar to those encountered with reductions. A simple method of implementing the multicast is to copy the message and send it separately to each element, allowing the standard message forwarding mechanism deal with migrating elements. Obviously, this likely results in several identical messages being sent between the same processors. The efficient way to implement multicasts is to distribute the message to all processors (via span-tree, or some machine-specific broadcast mechanism), and then have the local multicast controller copy the message for each local element. As was the case with reductions, dealing with migrating elements complicates the algorithm. We devised two methods for ensuring that each migrating element receives the message once.

Since multicasts are initiated by a single element, not by all elements in the same order, having each processor keep track of a serial number as we implemented for reductions does not work. One method is to use serial numbers, but to serialize all multicasts through a single processor. Any multicast message first goes to the main multicast controller, which assigns a serial number and then sends the message to the other processors. Each object contains the serial number of the last message it has received. Multicast messages are delivered in order, so each processor delivers the message to those objects who have not yet received it, but have received the previous multicast message. Then any previously-buffered messages with the correct serial numbers are delivered. Finally, the multicast message must be buffered until every destination object has received it.

A straightforward method for determining when a particular multicast message has been delivered would be for each processor to send the multicast controller an acknowledgment of the delivery of the multicast message for each element. Each processor would first participate in a reduction tree, which would tell the controller how many destination objects received the message initially. Later, as in-transit objects arrive at processors, those processors would deliver the multicast and send another message to the controller acknowledging the delivery. As soon as the number of delivered messages matches the number of destination objects, the controller broadcasts a message to the processors telling them to free the multicast message. An alternative method is to have a low-priority systolic message moving up and down a spanning tree, with the multicast controller at its root. Periodically, the controller sends a message containing a list of undelivered multicast message, and a list of messages to be freed. Initially, the *to-free* list is empty. Each processor stores the *undelivered-message* list, frees any messages that are in the *to-free* list, and passes the message down the spanning tree. When the message gets to the leaf processors of the tree, each leaf processor returns a new message containing a counter for each undelivered message. A reduction is performed on the array of counters, so when the message returns to the multicast controller, the counters record the total number of objects which have received each message in the *undelivered-message* list. When a counter indicates that all receivers have received one of the undelivered messages, the multicast controller adds the message serial number to the next *to-free* list. The controller can initiate this process whenever it deems necessary, perhaps by monitoring the total size of the outstanding messages and initiating cleanup when the size exceeds some threshold. Messages traffic can be reduced further by piggy-backing the data on other outgoing multicast messages.

We devised a second method to implement multicasts which eliminates the possible bottleneck of the serializing processor, but requires additional data in the message. As described in the discussion of message forwarding, each processor has a map of where it each object recently resided. Using the information in that map, the originating processor assigns responsibility for delivering the message to each object to the processor where that object was recently located. Let N_r be the total number of receivers. A message is broadcast to each processor containing the multicast data and an array $P[0...N_r - 1]$, where $P[i]$ is the processor responsible for delivering the message to receiver i. Since the originating processor's knowledge of object locations may be out of date, object i may not be at processor $P[i]$, but the common case is that a local send will serve to deliver the message to object i. Occasionally, when an object is not at the predicted processor, $P[i]$ uses the forwarding mechanism to deliver the message on to object i.

Since only a relatively small number of message sends are multicast messages, we currently use the "inefficient" method of performing multicasts by explicitly sending a separate copy of the message to each destination object. However, if applications arise which perform frequent multicasts, either of the other methods can be added without any changes in the multicast interface.

2.2.3 Element Migration Mechanism

Element migration is made up of three independent components: an object movement mechanism that moves the data from one processor to another, a message forwarding mechanism, which directs messages to a migrated object, or buffers messages for objects currently in transit, and the load balancer, which decides when to migrate elements, and to where they should migrate.

Object movement is accomplished by calling the `migrate(destination)` method on a particular array element. The element calls its element manager, which allocates a migration message large enough to hold the element data. Then the manager moves the element data into the migration message by calling the user-defined `pack()` method. Once the migration message reaches the destination processor, the *migration constructor* for the element type is called, which unpacks the element state at the destination. The last action of the migration constructor is to inform the local array manager branch that the object is now established at the destination processor and ready to respond to remote invocations.

2.3 Performance

We devised an experiment to determine the cost of message forwarding, and the results are shown in Figure 2.8. The experiment was a ping-pong message program, in which a messages was repeatedly sent back and forth between two elements of an array. One element remained on a processor, while the other element was migrated at regular intervals. We measured the time to perform a number of message exchanges for various message sizes. Figure 2.8 also shows the performance of a Converse ping-pong program, which has no migration or forwarding overhead. The experiment was run on eight processors of a Cray T3E.

The top curve (labeled "Migrate = 1") shows the average time to send the message from one element to another when the second element migrates to a new processor after every message receive. Each message requires about three times longer than a raw Converse message send. This time corresponds to the extra messages required for forwarding. First, the message is sent to the migrating object's original processor. There, the message is forwarded to the new location of the object. When that message reaches the destination, the update message is sent back to the fixed object. Finally, the element receives the message and echoes it back to the first element. While that message is in transit, the migrating element moves to the next processor, resulting in additional traffic. This represents a worst-case situation for message forwarding, which is unlikely to occur in actual applications.

Successive curves represent ping-pong performance when migration is less frequent (every sec-

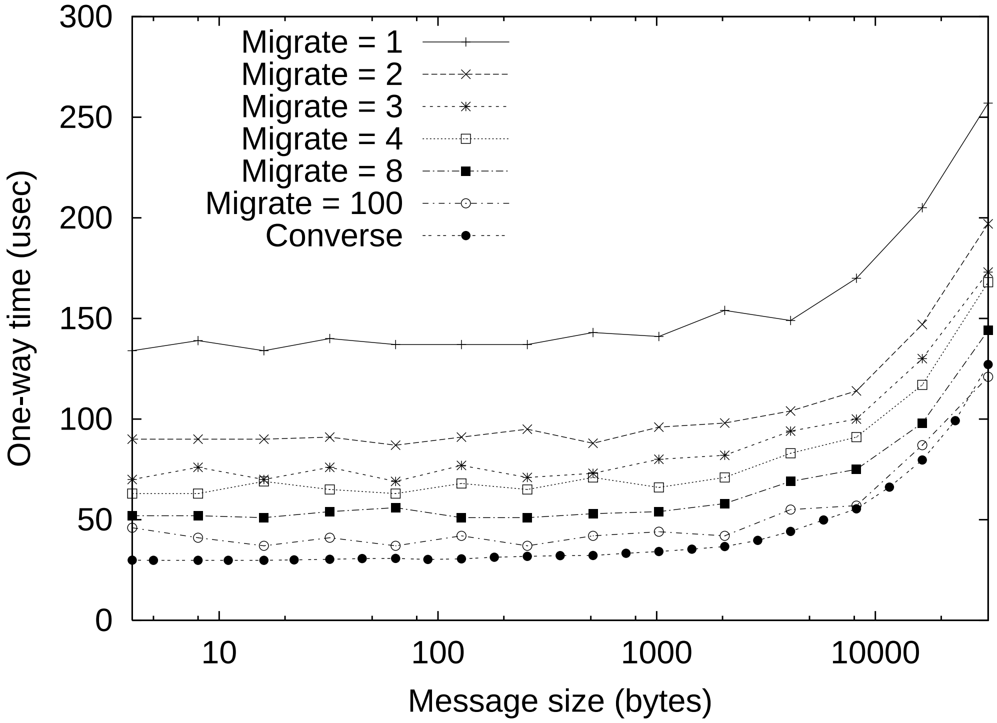

Figure 2.8 Message times for a ping-pong experiment with migration. A message is repeatedly sent between two elements of an array, while one of the elements periodically migrates to a new processor. Each curve shows the message time for different frequencies of migration (every 1, 2, 3, 4, 8, and 100 ping-pong cycles). The bottom curve shows the raw performance for a Converse-level ping-pong benchmark.

ond, third, fourth, eighth and hundredth cycle). As expected, less frequent migration leads to diminished impact on performance. In the last case, where the element migrates only after receiving one hundred messages, the performance with migration is the same as that of a Converse ping-pong benchmark, indicating that the overhead of the array layer is negligible as long as migration is not too frequent. In fact, this represents the typical case in applications, where elements only migrate occasionally, and process hundreds of messages before relocating.

CHAPTER 3

LOAD BALANCING AND APPLICATIONS

Parallel applications on popular computing platforms experience load imbalance from both external and internal (application-induced) causes. The following sections discusses these problems in more detail.

3.1 Background Load

A large fraction of parallel applications today are loosely synchronous: computations on one processor continually depend on data produced by other processors.[1] Consider a molecular dynamics application which divides the simulation volume into cells distributed among various processors. Computation for each cell may depend only on data from adjacent cells during a single timestep. However, over a period of several steps, the data from a particular cell indirectly affects every other cell in the simulation. The dependency graph insures that no cell can proceed more than one step ahead of the slowest cell in the simulation. Some simulations require collective operations, such as reductions to compute global scaling factors, and in those cases the processors are even more tightly coupled.

In case of such globally synchronous computations, utilizing a shared workstation cluster presents special problems. Consider an application running on eight workstations of a cluster. When the computation begins, all eight workstations are otherwise idle. Let us further assume that our ap-

[1]Not all computations are globally synchronous. Divide and conquer computations, and state-space search computations lead to computations trees, where the only dependencies are between parent and child nodes of the tree.

plication can utilize all eight workstations with one hundred percent efficiency. Sometime during the parallel run, another user starts a single processor job (it may be as small as a compilation or a long-running single-processor computational job). Now our parallel job is receiving only about half of the processing power available on this workstation, but the other seven workstations are still fully committed to the parallel job. Yet, due to the dependencies among sub-computations, all the other processors will have to waste half of their time waiting for results from the busy processor. Thus, even though we have lost only 1/16th of the total computational power to the outside job, the performance of the program drops by fifty percent.

We have observed this effect while running our parallel molecular dynamics application, described in Chapter 5. To study this phenomenon further, we implemented an artificial benchmarks using Charm++.

Our benchmark is a simple simulation. The computational domain is a two-dimensional square, which is divided into a uniform two-dimensional grid. The grid is divided into multiple cells, which are allocated among the available processors. The problem can be thought of as a heat convection problem in a flat square plate. Several regions of the plate are "heated" to maintain a predefined temperature. During each timestep, the temperatures at all the points in the grid are updated using a Jacobi relaxation scheme: each point on the grid is given the new value based on the average of all its neighboring values. The relaxation algorithm is repeated until a convergence criteria is met. To test the convergence criteria, a global reduction must be carried out to determine the maximum error. When the maximum error falls below a preset threshold, the iteration may stop, and simulation proceeds to the next timestep.

We ran the Jacobi benchmark on eight otherwise-idle HP 735/125 workstations communicating over a 100Mbps ATM network. About thirty seconds into the run, we started a computationally intensive job on one of the workstations. Figure 3.1 shows the impact on performance: prior to the onset of the new job, roughly ten iterations were executed per second. After the job starts, the number of iterations per second drops to about seven. Although the total available compute power decreased only six percent, throughput decreased thirty percent. A common optimization in this algorithm is to reduce the frequency for checking the convergence criterion, since even on dedicated machines, the data dependencies introduced by performing the reduction step every

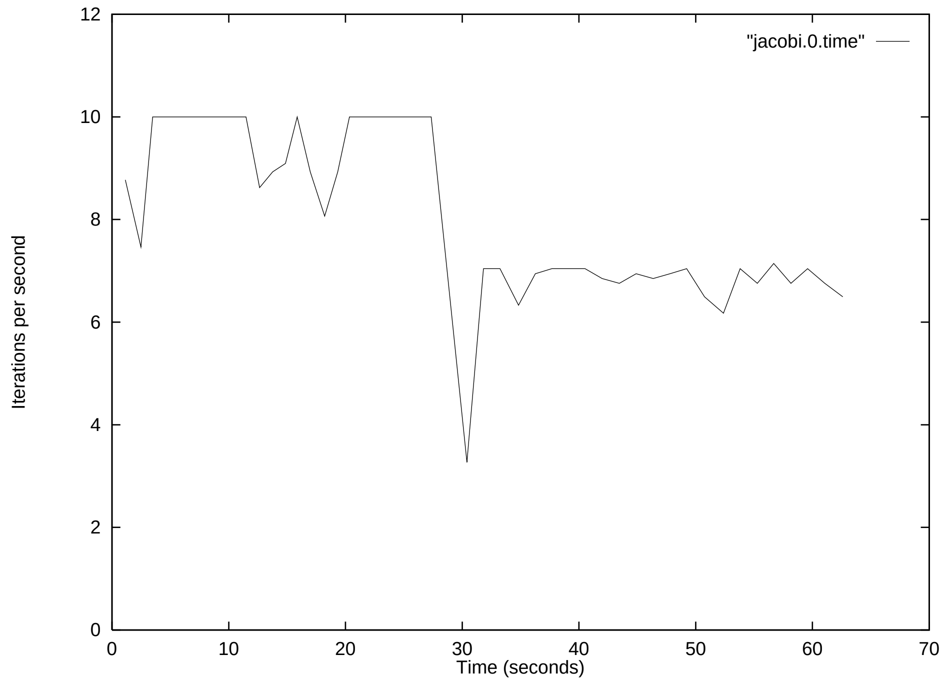

Figure 3.1 Number of iterations per second for the `jacobi` benchmark on eight processors. The cluster is composed of HP 735/125 workstations connected by an ATM network.

iteration severely compromises the parallel efficiency. When that optimization is applied to this benchmark, the application performs faster until the background task begins; then throughput drops by a full fifty percent. When the reduction does not occur, each neighbor B of an object A on the busy processor must wait twice as long for the slow object's data, and each neighbor of B must wait twice as long, and so on until all objects are penalized by the slowdown of A. The reduction masks this effect, because A sometimes can get a head start on other objects which are still waiting for the reduction to complete.

It is clear that we need some technique that allows us to utilize most of the available power. It is also clear, that this will require moving some of the work on the affected workstation to other processors. However, it is difficult to accomplish this without user intervention in traditional (e.g. MPI/PVM based) programs. A parallel object-oriented language, such as Charm++, provides the foundation for an automatic migration system.

3.2 Shared and Heterogeneous Clusters

Users may allow use of their desktop workstation only when they are not using it interactively. This is a special case of background load. Instead of adjusting a processor's load for optimal parallel performance, the parallel application must reduce its computation on the particular processor to an insignificant level. It may not be necessary to remove the process from the computer, as long as the CPU time consumed by the process is limited, perhaps to less than five percent of total time, and the memory use is reduced, either by causing the application to move data to other processors, or by allowing normal operating system procedures to move the now-inactive pages out of memory. In Charm++ programs, data is associated with objects rather than processors (there is typically little "home" node data), so moving those objects off a vacated processor will also reduce the memory usage of the process.

In the general sense, heterogeneous clusters are workstation clusters composed of machines with different processor architectures, data formats, and operating system environments. An important subclass that our work addresses is clusters of machines with similar processors, but different processing speeds. Workstation networks are much more likely to have a mix of processor types and speeds than dedicated parallel machines. As technology advances, new workstations likely to be faster than their predecessors purchased even just months earlier. Inter-processor communication costs also vary, depending on both network hardware in each computer, and the network infrastructure of the facility. Assuming fixed computation and communication costs when distributing a parallel task is bound to waste computational cycles unless individual machine characteristics are considered.

3.3 Application Imbalance

Application imbalance is a major contributor to inefficiency in all parallel programs. On a dedicated machine, it may be the only contributor, while on workstations it is only one of the factors which may have to be considered. Given an accurate model of computation and communication costs, it is possible to compute all possible work distributions for a given algorithm and select the one that gives best load balance. In practice, however, such a procedure is often too complex to solve exactly. Therefore load balancers for irregular application usually use heuristic methods to choose a "good" load distribution. With available processing power and communication characteristics changing continuously, the complexity of such heuristic load balancers increases, becoming both a programming and computational burden.

Initial load imbalance is a problem for the majority of parallel applications, whenever there is any irregularity in the structure of the computation. Most applications distribute work according to some heuristics based on the structure of the particular application. Our approach is to initially distribute objects according to some simple algorithm, run the application for a short time to collect object load statistics, and then to allow the general-purpose load balancing algorithm redistribute the objects according to the measured load. This approach is further examined in our discussion of molecular dynamics in Chapter 5.

Slowly-varying load occurs in many applications. Examples include molecular dynamics, where object loads change as the molecules move about the simulation space, or finite-element crack simulation, where increased computation is associated with the movement of the crack through the solid. The main tradeoff with slowly-varying load is how often to perform load balancing. If the load balancing procedure requires significant processing time, or if the load changes slowly, load balancing should be performed less often than when load balancing is fast or the load changes more quickly. Slowly-varying load in molecular dynamics is examined further in Chapter 5.

3.3.1 Sudden load changes

Sudden load changes can produce large decreases in parallel efficiency. Adaptive-refinement applications are one common class of programs which exhibit such problems. Consider an adaptive-refinement Jacobi algorithm, similar to the application described in Section 3.1. To the standard Jacobi algorithm, we add a refinement capability, so that the mesh can become several times finer in response to simulation conditions, resulting in a sudden increase in load for a few objects. Although the total load has increased, all of the load increase is concentrated on a few processors, resulting in poor load balance.

The Jacobi-refinement application was developed as part of an interactive web-based interface to parallel programs. The `Plate` simulation is of a metal plate with several hot spots, which simulates the cooling of the plate over time. Figure 3.2 shows the interface to the `Plate` program. In the upper-left corner is a false-color image of the current temperature of the plate. The upper-right corner shows the current processor assignment for the objects, where each object is responsible for a particular square region of the grid. The lower-left corner is a continuously-updated display of processor utilization, and the lower-right corner is a graph of processor utilization versus time. For demonstration purposes, both refinement and load balancing are triggered manually through the interface, rather than automatically as would occur in a real version of the application. The utilization versus time graph (lower right) shows the effect of refinement and load balancing. At program start, the application is well-balanced, since the mesh is uniform, and the objects are distributed evenly among the processors. At time interval 7, a small region of the mesh is refined, but the processor utilization drops to 77 percent. At step 16, load balancing is triggered. While load balancing and object migration is occurring, utilization temporarily drops to about 40 percent, however, a few seconds later, utilization rises again to about 92 percent.

3.4 Load Balancing for Finite Element Methods

Finite element methods (FEMs) are used to determine the forces which develop inside of objects in response to external forces. By studying stresses within the objects, researchers can determine how the object will deform. Finite element methods treat objects as a set of polygonal or polyhedral elements, which completely fill the object. The vertices of these elements are called nodes. Two methods are typically used to solve finite element problems. Explicit methods view the interactions among elements as a large set of simultaneous equations, which are solved using sparse matrix techniques. Implicit methods view the node points as discrete masses, which are minutely displaced due to external forces and interactions with the connected elements. The solution to the problem is found by repeatedly solving the equations of motion for the nodes until the node displacement stop changing. Either approach exhibits an irregular computational structure requiring load balancing for efficient parallel execution.

The application we studied is a finite element program used to study crack propagation, developed by Geubelle, Baylor, and Breitenfeld [52]. The non-linear behavior of a crack (the fact that two nodes on opposite sides of a crack will never exert forces on each other as they did before the

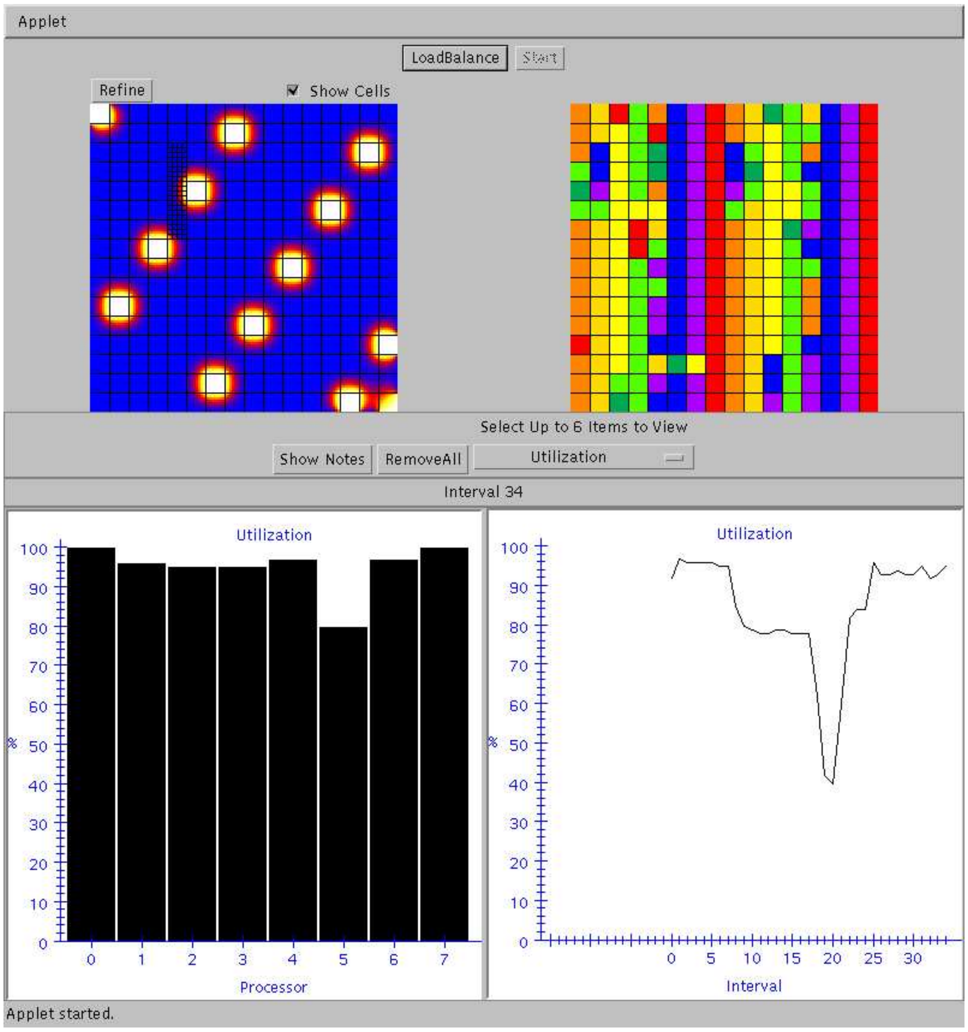

Figure 3.2 The web interface for the `Plate` application, showing refinement followed by load balancing.

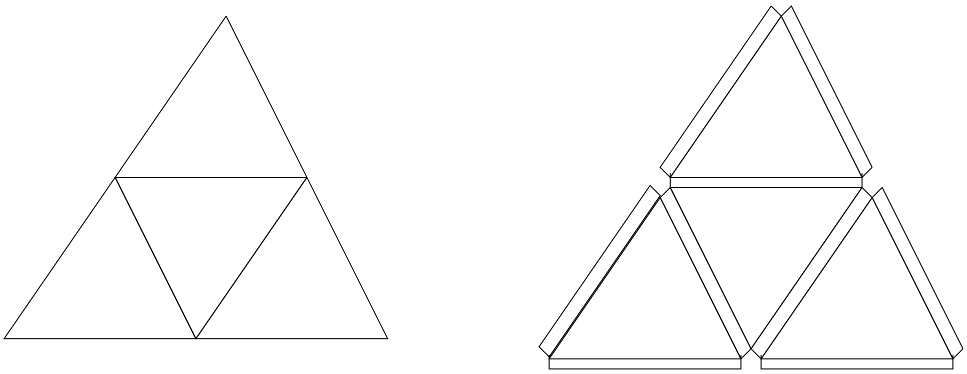

Figure 3.3 Left: Four elements in a typical finite-element mesh. The vertices of the triangles are the nodes. Right: The corresponding cohesive-element mesh, showing rectangular cohesive elements between each of the original non-cohesive elements.

crack formed) demand different mathematical treatment than standard FEMs. In this application, this is accomplished by using two types of elements, *non-cohesive* and *cohesive* elements. Figure 3.3 shows the two element types. Non-cohesive elements correspond to the elements found in standard FEM applications. Cohesive elements are elements which model the non-linear behavior of cracks. Between each pair of normal elements, there is a cohesive element. However, the initial, unstressed cohesive element occupies no space; the nodes of its adjacent non-cohesive elements are in the same positions. As forces are applied, the cohesive element nodes may move apart a short distance. If the nodes continue to move apart, the cohesive element "breaks", and forces are no longer transmitted from one node to another through the cohesive element.

Accurately modeling the behavior of the crack requires a much finer mesh of elements than is necessary to accurately model the transmission of forces through regions where no crack forms. Without foreknowledge of where the crack will go, the conservative solution would be to use a fine mesh throughout the material, producing a large amount of computation. The initial optimization that is used is to observe that, although we do not know exactly where cracks will form, empirical methods can establish regions where cracks are unlikely, so a coarser mesh without cohesive elements can be employed in those regions. A way to dramatically shrink the region requiring a fine mesh is to take advantage of the fact that after the initial formation of cracks, the initial cracks will propagate and possibly branch into more cracks, but unconnected new cracks are unlikely to form. Therefore, refining the mesh (and adding cohesive elements) only in the region of the crack reduces the computational load, but turns load balancing from a static into a dynamic problem.

3.4.1 The CohesiveFEM Application

The cohesive-element application we studied was a 2-D Fortran 90/MPI program which runs on the SGI Origin 2000. It uses the coarse-mesh region optimization, and the original developers are working on implementing the dynamic refinement. Our goal was to port the program to the Charm++ system, and to employ migratable objects for load balancing.

Porting to Charm++

The structure of the original program is described in Figure 3.4. Porting the program to Charm++ required several modifications:

1. Separate the Fortran computational routines from the MPI communication calls.

2. Write a message-driven Charm++ driver object to replace the Fortran main loop, to call the computational routines, and to store incoming messages for access by the computational routines.

3. Replace MPI communication calls with subroutines that call routines in the message-driven core.

4. Place all global variables in the Fortran program into a Fortran 90 derived type which could be allocated for each Charm++ object.

5. Change the driver objects (initially a Charm++ object group) into an object array.

Separating the Fortran computation from the communication was straightforward, mainly because all communication was performed at the top level of the program code. The procedure consisted of copying each block of computation between successive MPI_WAITALL() calls into separate subroutine, and replacing each block of code in the original program with a call to the new subroutine.

Once the code separation modifications were verified by comparing the program output to the output of the original code, a Charm++ main program was created to address steps two and three

```
begin
    Each processor reads in data
    Processors send data to neighbors      Neighbors are all processors
    Wait for neighbor data                 containing adjacent parts of
                                           the mesh
    Perform setup computations
    Send to neighbors
    Wait for neighbor data
    for i = 1 to num_steps
    begin
        Compute stage 1
        Send results to neighbors
        Wait for results from neighbors
        Compute stage 2
        Send results to neighbors
        Wait for results from neighbors
        Compute stage 3
        Send results to neighbors
        Wait for results from neighbors
        Compute stage 4
        Send results to neighbors
        Wait for results from neighbors
    end
    Finish computation
end
```

Figure 3.4 The original structure of the CohesiveFEM application.

of the porting procedure. The structure of the Charm++ main program is shown in Figure 3.5. In addition to these functions which describe the flow of control of the program, there are `send()` and `recv()` calls, which replace `MPI_SEND()` and `MPI_RECV()` in the Fortran code. The replacement send call packs the Fortran data into a Charm++ message and sends the message to the next stage of the neighbor objects specified by the Fortran code. The `recv` call stores the buffer pointer provided by the Fortran routine, and keeps a count of how many messages are expected. The next stage executes only when all the expected data has arrived and been copied into the specified Fortran buffers.

The third step of the porting procedure required turning the processor state into object state, so that multiple objects (which we named "chunks") could coexist on a single processor. The original code stored all program data in a number of global variables and arrays. These were all moved into a Fortran 90 derived type. At program start, the Fortran code allocates one of these chunk data structures, and returns a pointer to the structure to the Charm++ routine. This pointer is then passed back as a parameter to each call to the Fortran subroutines. Then, each variable access in the Fortran code is replaced with a reference to a field in the chunk data structure. For example, where the original code might access `X`, now it accesses `chunk%X`. Although this required changes to almost every line of the original code, the changes were simple, and were partially automated using editor macros.

At this point we have a Charm++ program that reads the same data files as the original program and produces identical output. The original program used a preprocessor to generate separate input files for each processor, so running on N_p processors required a set of input files which would not work unless exactly N_p processors were used. By allowing multiple objects per processor, (or even no objects on some processors), the Charm++ version of the program will run any input file set on any available number of processors. It seems reasonable to expect that dividing the work into smaller pieces would impose a performance penalty, due to factors such as additional loop overhead, but, as will be seen in the next section, there are advantages which offset these penalties.

```
class Cohesive
begin
    method Cohesive
    begin                                     The class constructor
        Call Fortran READDATA() routine       Sends data to neighbors' Cohe-
                                              sive::Setup
    end

    method Setup
    begin
        Buffer incoming message
        if all neighbor messages arrived then
            Call Fortran setup routine        Sends data to neighbors' Cohe-
                                              sive::Stage1
            i ← 1
        endif
    end

    method Stage1
    begin
        Buffer incoming message
        if all neighbor messages arrived then
            Call Fortran STAGE1() routine     Send data to neighbors' Cohe-
                                              sive::Stage2
        endif
    end

    method Stage2                             Omitted for brevity, but
    method Stage3                             similar to Cohesive::Stage1

    method Stage4
    begin
        Buffer incoming message
        if all neighbor messages arrived then
            Call Fortran STAGE4() routine     If i ≤ num_steps, sends data
                                              to neighbors' Cohesive::Stage1
            i ← i + 1
            if i > num_steps then
                Call Fortran FINISHCOMPUTATION()
            endif
        endif
    end
end class
```

Figure 3.5 The Charm++ driver program for the CohesiveFEM application.

Load Balancing Issues

The cohesive element FEM algorithm presents several static load balancing challenges. First, since it uses an irregular mesh, a simple partitioning of the object will not produce efficient performance. Instead, load balancing must balance the work produced for the nodes assigned to each processors (or chunk), and the quantity of communication of that partition with its neighbors. Secondly,in contrast to standard finite-element problems, the cohesive-element algorithm requires more computation for a cohesive node than for non-cohesive nodes, therefore balancing the number of nodes alone is not an accurate way to balance the load. Although graph-partitioning algorithms exist which consider node and edge weights, using them also requires determining the relative computational costs of cohesive and non-cohesive nodes for each target computer architecture.

In the original code, static load balancing is performed by a preprocessor, which calls the METIS graph partitioning library [46], [53] to efficiently partition the finite element mesh. The preprocessor does not know the relative cost of cohesive and non-cohesive nodes. Hence, it cannot pass node costs to METIS, which would allow METIS to balance node loads rather than node counts. Instead, it partitions the cohesive and non-cohesive regions separately, and gives each processor one cohesive partition and one non-cohesive partition. This procedure does balance the load, but at some increase in communication costs, since the preprocessor has no way to insure that a particular processor's cohesive and non-cohesive partitions are contiguous.

Since the Charm++ version of the program uses object arrays, it takes advantage of the load balancers described in Chapter 6. However, the preprocessor is still necessary to produce an input file for each chunk. The Charm++ load balancers are all designed assuming that there will be many objects per processor, so the preprocessor is used to generate several times more partitions than the maximum number of processors to be used. Although METIS ensures that each object has approximately the same computational load, it is not aware that groups of objects will be running on the same processor. Therefore, METIS cannot suggest processor assignments for the objects, nor can it dynamically account for changing machine characteristics due to background load. Also, since METIS is only used in the preprocessor stage, it cannot react to changing object load due to mesh refinement.

Figure 3.6 demonstrates an interesting effect of grainsize on application performance. After

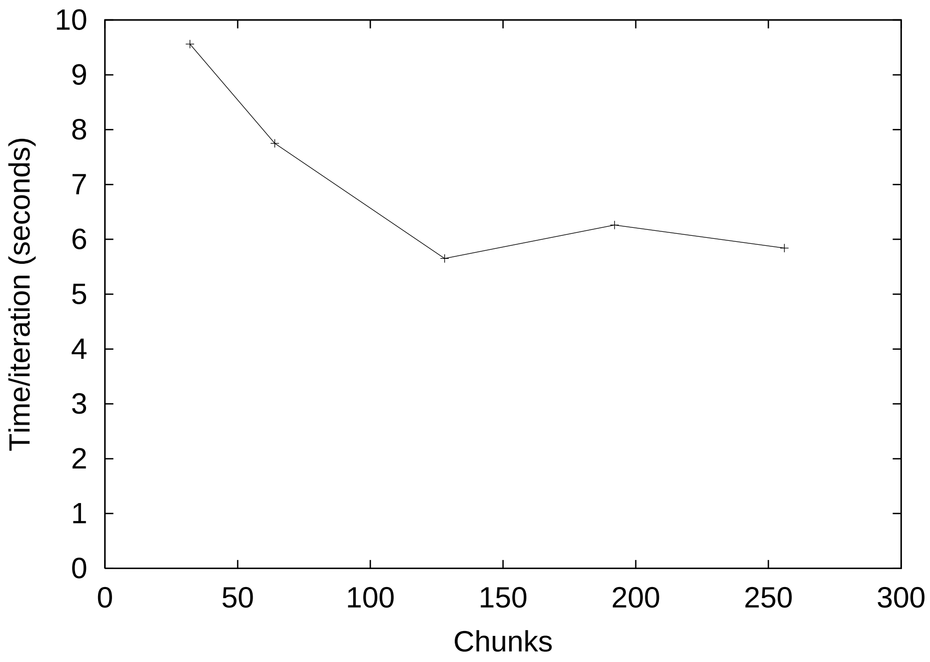

Figure 3.6 Execution time (seconds/iteration) for CohesiveFEM for different numbers of chunks on 32 processors

porting the original application to Charm++, we wanted to determine how much performance overhead results from using multiple chunks per processor. Since the cohesive mesh refinement is not yet implemented the METIS preprocessor produces constant-sized chunks, and the initial round-robin distribution of chunks to processors produces good load balance. We naively expected that, assuming METIS had initially produced balanced chunks, using one chunk per processor would yield the best performance, but that subdividing the mesh into more objects would yield only a slight increase in execution time, due to extra communication overhead. The experiment we designed was to partition a 600,000 node finite element problem into a range of partition sizes, allowing runs with 32 to 256 chunks. We then ran each case on 32 processors on the Origin 2000.

Surprisingly, best performance resulted for 128 chunks (4,700 nodes per chunk), or 4 chunks per processor. One reason for improved performance may involve cache performance. Since smaller objects have smaller data arrays, those arrays may fit entirely into cache. In contrast, large objects would not be able to hold all their data in cache at once, resulting in increased cache misses. In many applications, the compiler can split the loops into pieces which can be computed completely in cache; however, this implementation uses indirection in its array accesses which make such automatic optimizations unlikely. Alternatively, the authors of the original program might have hand-optimized the loops, either to fit in cache or to permit automatic optimization, but they chose not to. Another reason for improved performance with more objects may be that multiple objects per processor permit pipelining of computation and communication. As soon as one chunk has data available, it begins computing, even though the others may still be waiting for data. In the single-chunk per processor case, each processor must wait for all of the neighbor data to arrive before computation can start. Of course, for load balancing purposes, good load balance is more easily achieved with more objects per processor.

CHAPTER 4

THE LOAD BALANCING FRAMEWORK

We have designed a framework for creating automatically load balanced Charm++ applications. The load balancing framework works well for a variety of applications, without requiring application-specific customization of the load balancing strategy. Furthermore, we have gone to some length to insure that other programming models, (migratable threads, for example) may also take advantage of the framework.

Two main observations make possible this unified approach to load balancing. First, although an application's load depends on various factors, time-based load measurement is a general way of analyzing the load in any application. Second, viewing an application as a network of communicating, migratable objects permits separation of the decision-making process of load balancing from the application-specific mechanisms of work movement.

4.1 Requirement for a Load Balancing Framework

Several requirements guided the design of the general-purpose load balancing framework. First, to be independent of any particular application domain, we had to select some universal way of measuring load. We satisfy this requirement by using object execution time. Since considering communication is also critical for measuring program performance, the framework also tracks communication between processors. We assume that objects will be fine-grained enough that load could be controlled by migrating entire objects among processors, so that the framework does not have to provide a capability for objects to transfer work among themselves. Experience with writing migratable objects proved that creating objects that could pack and unpack their state at any time during their execution was complex. Packing and unpacking is greatly simplified if objects are asked to migrate only at certain points in their execution. Therefore, the framework had to provide synchronization facilities so that load balancing is activated only when objects are prepared to migrate.

The framework was designed to simplify experimentation with various load balancing strategies and various applications. Achieving this required the framework to implement a common API for interactions between applications and load balancing strategies. An application should only make calls to the framework, not directly to the load balancing strategy, so that an application would not need to know which load balancing strategy was in use. Similarly, the load balancing strategy only interacts with the framework, so that it could function with any application.

4.2 Overview

Figure 4.1 shows the main components of an application using the load balancing framework. Note that applications and strategies interact only with the *load coordinator* component of the framework, not with each other, and that the coordinator layer does not use inter-processor communication.

4.2.1 Terminology

In this section, we will define some terms that will be used in the description of the load balancing framework.

Load Coordinator: The load coordinator is the component of the framework that provides the interface between applications and load balancing strategies. The coordinator uses the load database to keep track of objects and communication history, and helps coordinate the migration of objects.

Load Database: The load database is a component of the load coordinator that stores the execution data.

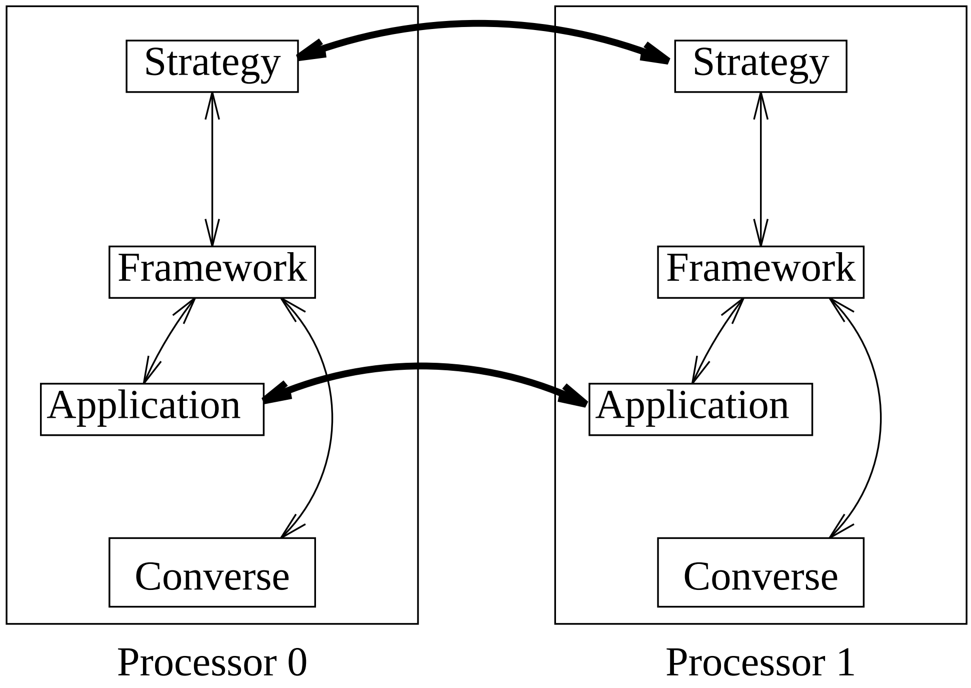

Figure 4.1 The components and interactions in the load balancing framework on two processors. Bold lines indicate off-processor communication.

Load Balancing Strategy: A load balancing strategy is the component of the framework that analyzes the data stored by the load coordinator and makes the decisions about where objects should migrate.

Object: An object is a piece of an application which consumes processor time, and communicates with other objects.

Object Manager: An object manager is a component, typically provided by the language runtime, which customizes framework interactions, so that the application objects in most cases need not interact directly with the framework, or interact in ways which fit the particular type of object (so that, for example, object array elements interact differently than migratable threads). An object manager is typically a parallel object, with a representative on every processor.

Object ID: An object ID is an identifier provided to the framework by the object manager that, along with the object manager ID, uniquely identifies a particular object, no matter to what processor the object moves.

Object Manager ID: An object manager ID is a unique identifier that each object manager provides to the framework. On each processor, the local object manager representative provides the same object manager ID, so that references to object manager IDs are valid on any processor.

Object Manager Handle: When the object manager is created, each representative registers itself by providing its object manager ID to the load coordinator. The coordinator returns an object manager handle which is used in future calls to the coordinator for more efficient identification. Object manager handles are only valid on the processor where they were created, whereas object manager IDs are valid everywhere. The coordinator can retrieve the object manager ID from the object manager handle, so when an object manager handle is available, it is not necessary to also refer to the the object manager ID.

Object Handle: When an object is created or migrates to a new location, its object manager registers the object and provides the object ID to the load coordinator, and the load coordinator returns an object handle. The object handle identifies the object to the coordinator more efficiently, but is not valid across processors, and changes whenever the object migrates. The coordinator can retrieve the object manager ID and object ID from the object handle, so when an object handle is available, the IDs are not necessary.

4.2.2 Framework Overview

Figure 4.2 is a more detailed view of the program components on a single processor. Application objects are controlled by object managers; for example, each Charm++ object array instance has an array manager which serves as that array's object manager.

At the top level of the figure is the load balancing strategy. The coordinator periodically notifies the strategy of opportunities to perform load balancing. Upon notification, each local branch of the strategy may retrieve information from the local coordinator database about the current state of the processor and the objects currently assigned to it. Two types of information are locally available from the coordinator. The strategy may ask for machine information, such as total wall-clock time and CPU time consumed since the last load balance notification, how much time was spent idle, and an estimate of processor speed. The other data available to the strategy

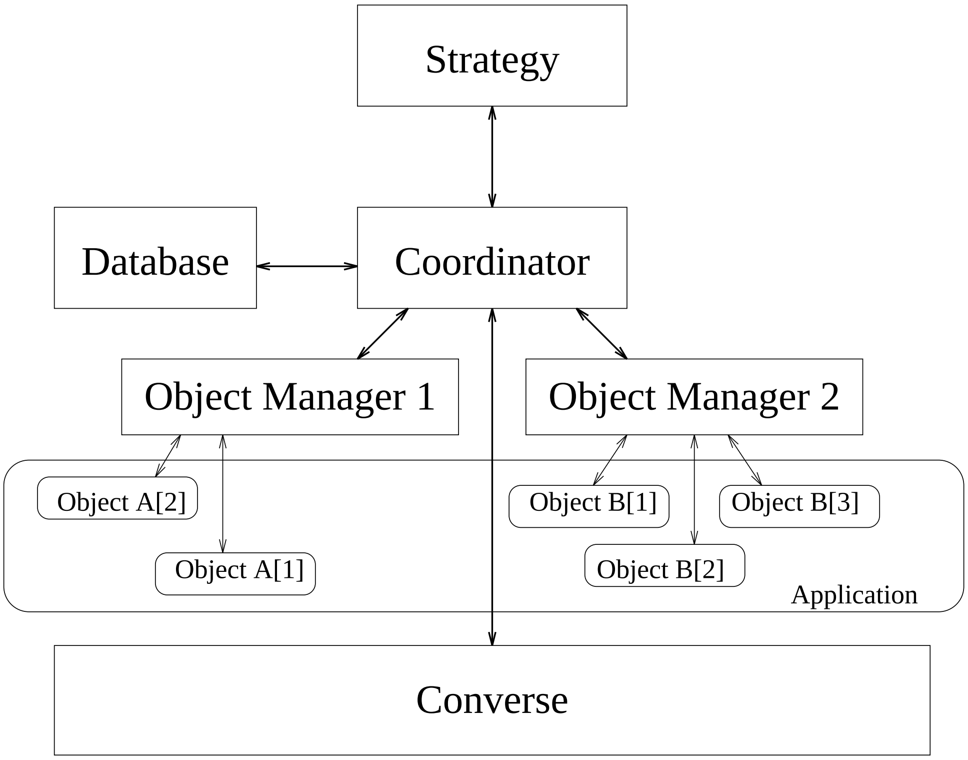

Figure 4.2 The components and interactions in the load balancing framework on a single processor.

is the locally-collected portion of the object-communication graph, including the processor time consumed by objects currently assigned to this processor (the nodes of the object-communication graph), and which objects sent data to other objects, and how much data they sent (the edges of the object-communication graph). Strategies determine when and where to migrate objects, and provide this information to the coordinator, which supervises the movements of the objects, informing the strategy as objects arrive. When the moves are complete, the strategy signals the coordinator to resume the objects.

The load coordinator interacts with objects through a number of object managers. For example, for object arrays, each array manager is an object manager. Object managers are responsible for reporting the arrival and departure of objects to the load coordinator. They also report when a particular object is being executed, so that the coordinator may accumulate time for the method, and when objects send messages to other objects. Object managers may register non-migratable objects, so that communication with these objects may also be taken into account. When the load coordinator receives a request to migrate an object, it forwards the request to the object manager, which coordinates the migration with the user object.

Since the application code is part of the objects, programming convenience demands that objects do not contain explicit calls to the coordinator. Most coordination with the framework is handled by the object manager. The interface between objects and object managers is specific to the object type. For Charm++ objects, for example, objects may provide `pack()` and `unpack()` methods, which the manager would call to perform migration, functionality that might be unnecessary for a module using migratable threads, whose managers would always move the entire stack.

One important 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 object manager of potential migration points through an *at-sync* call, which in turn informs the load coordinator. Those objects cannot leave the migratable state until the coordinator signals the object manager to resume them. For object arrays, at regular intervals each object calls an `AtSync()` method of the array manager, and perform no further computation until the array manager invokes their `ResumeFromSync()` method. Before `ResumeFromSync()` is called, the framework may or may not request that the object migrate to a new location. Currently, the `AtSync()` call acts as a *local barrier*. A local barrier is a barrier only requiring the synchronization of objects on the current processor. The overhead of such a barrier is minimal, because off-processor communication is not required. Furthermore, until all objects have reached the barrier, it is likely that objects not at the barrier still have computation to perform, so processor time is not wasted. Once all synchronized objects on the processor have reached the barrier, the coordinator triggers the strategy, which at that point may decide to begin a load balancing cycle.

4.3 Framework Details

The following sections describe in more detail how various components interact with each other within the framework, and the responsibilities those components are expected to meet.

4.3.1 Object Managers

Object managers provide the "glue" between various programming models which take advantage of the framework and the rest of the framework. Object managers are responsible for four main activities: object registration, data collection, managing synchronized objects, and coordinating migration.

Object managers identify themselves, and their objects, to the framework at run-time by making registration calls to the load coordinator. The framework expects object managers to be a parallel objects, in that each manager has a component (or branch) on every processor, but those components identify themselves by a common name. When an object manager is created, it must select a unique object manager ID. Each branch of the manager first registers itself with the local coordinator, obtaining an object manager handle from the coordinator. At registration time, the object manager supplies a callback function so that the coordinator may pass along migration requests. After each branch registers itself, the manager branch may begin creating and registering the objects which will initially reside on that processor.

The object manager is also responsible for reporting object execution and communication to the load coordinator. Just before code for an object is executed, the object manager reports the object start to the coordinator, so that the coordinator may record the object's start time. Similarly, when the object stops executing, the object manager reports to the coordinator, so the coordinator may store the accumulated time consumed by the object. The object managers must also report message traffic to the coordinator. Whenever a object is running (as reported to the coordinator by its object manager), the handle of the currently-running object is stored by the coordinator. When that object sends a message to another object, the sender reports the send to the local representative of the receiver's object manager. The receiver object's manager reports the object manager ID and object ID of the receiver and the number of bytes sent to the coordinator, and the coordinator adds the sender handle of the currently-running object. If no object is currently running, the coordinator assumes that the message was received from some entity not registered with it, and records the communication as a processor-to-object message.

As described previously, it is often convenient for application programmers to only concern themselves with migration when the objects are in certain points in their execution. Such points are indicated to the framework by the local barrier mechanism. At creation time, the object manager tells the coordinator whether or not the object is a synchronized object. Periodically, every synchronized objects will tell their object managers that they are at a synchronization point. The object manager passes that information to the coordinator. Later, after any migration is completed, the coordinator will pass a resume message from the load balancing strategy to the manager. The local manager then resumes all the synchronized objects on its processor. How different object managers accomplish this depends on the design of the language run-time. For Charm++ objects, each object tells its manager whether it is a synchronized object at creation time. At the synchronization point, the object calls an `AtSync()` method. To resume the object, the manager calls the `ResumeFromSync()` method of the object.

Coordinating object migration is also the responsibility of the object manager. When the object manager registers itself, it provides a callback function pointer for the coordinator to use to request that an object move to a new processor. When objects are created and registered with the coordinator, the manager indicates whether each is eligible for migration. To move an object to a new processor, the coordinator calls the migrate callback for the object manager branch of the processor where the migrating object currently resides. The object manager unregisters the migrating object (and if it was a synchronized object, also removes it as a local barrier client), and packs it (or asks the object to pack itself), and sends the object to the object manager on the destination processor. When the object arrives at its new destination, the manager registers the object at its new location, obtaining a new object handle for the object. If necessary, it registers the object as a local barrier client, and then unpacks the object. Once the object is unpacked and ready to resume execution, the object manager informs the coordinator that the migration is complete.

4.3.2 Objects

As described above, the framework design minimizes the required interactions with objects. Most information about objects is provided by the object manager. For non-synchronized objects, the framework imposes no responsibilities on the object, although, depending on the object type (array element, thread, etc.), the object manager may require information in order to fulfill its responsibilities. For synchronized objects, the only requirement the framework expects objects to satisfy is to activate the at-sync call "often enough." Since the frequency of the trigger determines the frequency of opportunities for load balancing, the factors which determine how often load balancing is required provide a bound for how often the objects must trigger the at-sync. Some of those factors are application-specific, so it is reasonable to let the application have control over how often load balancing can occur; however, when attempting to react to external effects on workstations, load balancers may need to act more often than application factors alone warrant.

4.3.3 Strategies

Strategies implement the "intelligence" of the load balancer, therefore various strategies differ widely. An application will have only one strategy active, but many different strategies exist for different machines or types of applications. We will discuss different strategies in Chapter 6, but here we will focus on the responsibilities of strategies within the framework and the strategy-coordinator interface.

A strategy does not generally need to register itself to the load coordinator, but if it does not, it will not be able to migrate synchronized objects. There are two callbacks which the strategy may register with the coordinator to receive notification of events. First, a strategy can register a callback so that the coordinator can notify it when all of the local synchronized objects have reached the local barrier. Second, in order to know when migration is complete, the strategy may register a callback to be notified by the coordinator whenever a migrating object arrives on the processor.

Commonly, a strategy will be implemented as a set of objects, with a representative on each processor. At creation, the branch on each processor will register itself to receive notification of local barriers and migrations. Then it will wait until the local objects reach the local barrier.

When the local barrier callback is called, the strategy will decide whether load balancing should occur at this point. If load balancing is unnecessary, the strategy will tell the load coordinator to resume the local objects, and then wait for the next callback. If load balancing is necessary, each strategy branch will request data about the state of its processor, and that processor's objects. The object data is a list of statistics for each object on the processor, including the object ID, and how much wall and CPU time the object consumed since the statistics were last cleared. The strategy branch may also request a list of outgoing communication links, where each link represents message traffic which occurred between two objects, an object and a processor, or two processors. The link statistics include how many messages were sent, and the cumulative number of bytes sent. Furthermore, the strategy may obtain processor-specific information, including the total (object and non-object) wall and CPU time used by the processor, the idle-time (time when there were no tasks in the Converse scheduler queue), overhead load (time spent by the process, not associated with any of the objects), and processor speed (a relative speed-index, determined by the coordinator at startup by a simple computational benchmark).

Most strategies will send this data, or some portion of it, to other processors for analysis. For instance, object graph-based strategies will most likely collect the data on one processor so the entire object-communication graph can be assembled and migration decisions made centrally. Other strategies may have each branch send the data to a subset of the processors, and await similar data from those processors. In either case, the strategy decides what object to migrate, and where to migrate them. Each branch will then determine, or be informed, which objects it should migrate away, and how many incoming objects to expect. The strategy branch tells the coordinator which objects to move, and to where they should migrate. Meanwhile, as migrating objects from other processors arrive, the coordinator informs the strategy through the migration callback. When all outgoing objects have left, and all incoming objects are established, the strategy directs the coordinator to resume those synchronized objects now on its processor, while non-synchronized objects continue execution without further interaction. Then the strategy awaits the next local barrier.

4.3.4 The Load Coordinator

The load coordinator serves to connect object managers (and their associated objects) to load balancing strategies. To accomplish these functions, it implements an object manager interface and a strategy interface. The coordinator has several sub-parts which coordinate the interactions between applications and the load balancer, including:

Object registration: Identifying objects to the coordinator, and to strategies.

Data acquisition: Recording object computation and communication in the load database.

Data reporting: Reporting the acquired data to load balancing strategies.

Local barriers: A service to simplify migration, by controlling when migration can occur.

Migration: Passing migration commands from the strategy to the object managers, and passing migration status back to the strategy.

Since the interfaces were already described for each component, this section will omit those descriptions and focus on implementation details.

The load coordinator is a sequential library; each processor has its own load coordinator. For efficiency, the coordinators trusts the object managers to keep object manager IDs and object IDs consistent, rather than implementing global registration functions. Similarly, the coordinator assumes that the manager will provide a valid receiver ID when reporting a message send. Data reporting is also local; each processor maintains only a portion of the object-communication graph, namely, the list of objects currently on the processor, and the communication data for messages originating on the processor. This leaves open the option for a strategy to only analyze local data without incurring the penalty of constructing the entire object graph. Local barriers have no need for off-processor communication, and although migration is a process requiring communication, it is closely tied to the object type, so it is left to the object manager to handle, with the coordinator serving only as a conduit for information between the managers and the strategy.

Some complex coordination problems arise with migrating objects joining local barriers. Consider a situation where all objects have left a processor. Because load balancing is triggered by all objects calling `AtSync()`, if there are no synchronized objects on a processor, load balancing will never be activated. One solution to this problem is to always have a dummy synchronized object, which when resumed immediately (or after an appropriate delay) calls `AtSync()` again. Now suppose that the load balancing strategy has decided to migrate all objects currently on this processor away, but to migrate other objects to this processor. If all of the departing objects leave before any new objects arrive, the only remaining object will be the dummy object, which may trigger the `AtSync`, starting a new load balance cycle before the current one is complete. An algorithm which avoids these problems is to have the strategy on each processor know how many objects will be arriving and leaving. The coordinator notifies the strategy whenever an object arrives or leaves. Only when all migrations involving this processor are complete does the strategy resume the synchronized objects, including the dummy.

CHAPTER 5

CASE STUDY: NAMD

NAMD is a production-quality, parallel molecular dynamics program we have been developing for the last six years. The first major redesign of the program, NAMD 2, was undertaken to increase the scalability and modifiability of the program. Increased scalability required a redesign of the computational core to increase the number of independently-distributed pieces of work (ie. objects). That redesign resulted in objects with a wide range of computational demands, which produce improved scalability only when load balanced effectively. We discovered that dynamic, measurement-based load balancing was the best way to achieve high parallel efficiency. Recently, NAMD 2 has been revised to utilize the load balancing framework describe in Chapter 4. Material from this chapter is largely based on [54].

Computational molecular dynamics is aimed at studying the properties of biomolecular systems, and their dynamic interactions. As human understanding of biomolecules progresses, such computational simulations become increasingly important. In addition to their use in understanding basic biological processes, such simulations are used in rational drug design. As researchers have begun studying larger molecular systems, consisting of tens of thousands of atoms, (in contrast to much smaller systems of hundreds to a few thousands of atoms a few years ago), the computational complexity of such simulations has dramatically increased.

Although typical simulations may run for weeks, they consist of a large number of relatively small-grained steps. Each simulations step typically simulates the behavior of the molecular system for a few femtoseconds, so millions of such steps are required to generate several nanoseconds of simulation data required for understanding the underlying phenomena. As the computation involved in each timestep is relatively small, effective parallelization is correspondingly more difficult.

Although the bonds between atoms, and the forces due to them, have the greatest influence on the evolving structure of the molecular system, these forces do not constitute the largest computational component. The non-bonded forces, the van der Waals and electrostatic (Coulomb) forces between charged atoms consume a significant fraction of the computation time. As the non-bonded forces decrease as the square of interaction distance, a common approach taken in biomolecular simulations is to restrict the calculation of electrostatic forces within a certain radius around each atom (*cutoff* radius). Cutoff simulation efficiency is important even when full-range electrostatic forces are used, since it is common to perform several integration steps using only cutoff forces between each full-range integration step. The following discussion therefore focuses on cutoff simulations performance.

The following discussion describes our experience with load balancing for our molecular dynamics application, NAMD 2. NAMD 2 is a Charm++ program, but implements its own migration system rather than using Charm++ migratable objects. NAMD 2 was designed in 1996, prior to the design and implementation of the framework discussed in this thesis. However, the measurement-based load balancing and object-migration techniques developed in this thesis were critical in solving the load balancing problems exhibited by this application. Originally, NAMD used its own load balancing system, but we have recently modified NAMD to use the general load balancing framework with a NAMD-specific load balancing strategy. Using the general framework makes the NAMD code somewhat smaller and easier to understand, and also allows NAMD to use the general-purpose load balancers provided by the framework, in addition to its own specialized load balancer. This section discusses the application in some detail, and describes how the load balancing techniques were applied to it.

5.1 Molecular Dynamics and NAMD objectives

Molecular dynamics (MD) serves a pivotal role in inferring functions of biomolecules from their structures. The number of structures obtained via x-ray crystallography and other means has increased tremendously in the past several years. Researchers believe that molecular dynamics simulations are critical in translating structure information into mechanisms underlying biomolecular functions. Successful simulations of biomolecular systems can lead to a deeper understanding of basic biological processes and provide new insight to aid rational drug design.

A major hurdle to successful deployment of molecular dynamics for these purposes is the sheer computational intensity of the required simulations. The biomolecular systems of interest today consist of ten thousand to one hundred thousand atoms. Most often, the dynamics of such systems must be studied over several nanoseconds to develop significant understanding of the phenomena being studied. At the same time, the time scale of atomic interactions requires that one simulates their behavior in time steps as small as one femtosecond (10^{-15} seconds). In spite of the dramatically increased speeds of individual processors, the number of computational steps required to complete relevant simulations is prohibitive on any single-processor computer.

Parallel computing provides the potential for making the required large-scale simulations feasible. Parallel machines with tens, hundreds, and even thousands of processors are commercially available now. Such machines provide the potential to reduce the simulation time from years to mere days, thus making it possible to investigate functional properties of structures identified by contemporary scientific studies in a timely fashion.

To realize this potential of parallel computers, it is necessary to develop an efficient parallel molecular dynamics program. Such a program must be scalable, so that it can exploit newer generations of parallel machines with thousands of processors efficiently. Although the overall computational complexity of simulations is very large, the complexity results from the large number of time steps required. Each time step itself is relatively small. Since each time step must complete before the next one can begin, there is little opportunity to parallelize across time steps, requiring extensive parallelization of computation within each timestep. For example, simulating a biomolecular system with 3762 atoms sequentially required only about 1 second for a single time step in a particular simulation. To run this computation effectively on a large parallel machine implies that each time step must complete within tens of milliseconds. Given the communication requirements of molecular dynamics, and the relatively non-homogeneous nature of its computations, this becomes a challenging feat.

The program described in this chapter, NAMD 2, is one of the new parallel programs aimed at utilizing large parallel machines in a scalable manner. The nonbonded force computations, which constitute the dominant component of the overall computational cost, require calculation of pairwise interactions between atoms. In most common methods, a cutoff distance is used. Nonbonded interactions between atoms beyond this cutoff radius are either not calculated or calculated less often. This geometry-dependent nature of the computation complicates parallelization strategies for molecular dynamics programs. As described in Section 5.2, NAMD 2 uses a unique decomposition strategy that combines the advantages of spatial decomposition and force decomposition, which permits the program to utilize a large number of processors.

In addition to speed, maintainability and extensibility are also required in a parallel molecular dynamics program. Scientific studies of biomolecular systems often involve novel approaches. For example, one may be interested in steering small groups of atoms through a molecular system to study the forces experienced by them. Studies of specific systems may require periodic or non-periodic boundary conditions. Alternative numerical integrators are employed for increased accuracy or speed. A well-motivated application-domain programmer should be able to extend the parallel program to permit such experimentation. This requires a highly modular organization for the program. Ensuring such modularity is more difficult in a parallel program than a sequential one, because the new algorithm must correctly interleave the parallel communication logic. NAMD 2 handles this problem in part by providing the infrastructure (class hierarchy) that embodies commonly needed molecular dynamic mechanisms. Parts of this infrastructure include generic force computation objects along with *patch* (see Section 5.3.1) objects representing cubical regions of space. A numerical integration scheme, for example, can be written solely from the point of view of all the atoms within this cubical region, ignoring all other aspects of parallel communication. A scheme for maintaining *proxy* patches on each processor ensures that data communication is not visible to other components of the program. The generic force computation objects make it easy to add new types of forces, such as those required in steered molecular dynamics or free energy calculations. This modular organization is further enhanced by using C++, a language that encourages modularity, and Converse, a portable parallel run-time framework that allows multiple parallel paradigms to co-exist in the single application. As a result, each module can be written in a parallel paradigm most appropriate to it. This extensibility-oriented design is elaborated on in Section 5.3, which also further explains the parallelization strategy, and the specific algorithms used in various components of NAMD 2.

Although the parallelization strategy enables scalability, a good load balancing scheme is necessary to achieve good performance. The use of adaptive measurement-based periodic load balancing in NAMD 2 is effective in achieving good performance even for difficult MD simulations (*e.g.* those involving non-periodic configurations with high variations in atom density). This strategy and the results of some performance evaluation studies are discussed in Section 5.4.

5.2 Parallelization Approaches for Molecular Dynamics

The molecular dynamics computation involves calculating forces on all atoms during each time step, and integrating the forces to update the positions and velocities of the atoms. The integration requires a comparatively small fraction of time, and is a completely local operation. The force computations can be broadly divided into two categories: bonded force computations and non-bonded force computations. There are several categories of bonded forces, yet they all involve between two and four nearby (bonded) atoms. Non-bonded forces are due to Lennard-Jones and electrostatic interactions between atoms. Non-bonded forces decrease as the distance between atoms increases. Also, any given (sufficiently large) region of space is likely to be charge-neutral, thus reducing its effect on distant atoms. As a result, many simulations use a cutoff radius to save computation time: non-bonded interactions between atoms beyond this distance are not considered. For some simulations, the cumulative effect of far-away interactions cannot be ignored. Even in such simulations, a cutoff distance can be used to reduce computation: Forces due to atoms beyond the cutoff distance vary relatively slowly, and therefore can be computed less often, typically only once every 2-5 steps. It is therefore appropriate to focus our attention on cutoff based simulations for the purpose of discussing alternate decomposition strategies.

The non-bonded computations constitute between 80 and 95 percent of the overall computation, depending on the cutoff radius used. (When full-range interactions are included, one can often use a cutoff radius as small as 8Åfor the cutoff-only steps, compared with 12 to 18Å used in cutoff-only simulations.) Although the non-bonded force computations are dominant, one cannot ignore the computational cost of bonded forces in designing the decomposition strategy. For example, if one were to completely sequentialize their (say five to twenty percent) computations, the overall speed up would-be limited to at most twenty. We aim at making the program *scalable* to a larger number of processors.

Given a simulation of fixed size, there is naturally a limit to the number of processors one can use to achieve higher performance, i.e., no algorithm is continuously scalable. Instead, we use the following definition of scalability, based on [55] and [56]. A parallel algorithm is said to be *isoefficiently scalable* if one can increase the number of processors used by it (P), and still retain its parallel efficiency by increasing the size of the problem being solved. (Thus, using a scalable algorithm, we will be able to solve larger problems with a larger number of processors).

The parallel efficiency η is defined through the following equation:

$$\eta = \frac{\textit{sequential time}}{P \times \textit{parallel time}} = \frac{T_s}{P \times T_P}$$

For a solving a particular problem, when the number of processors is increased, the efficiency will normally decrease (due to increase in communication costs, for example). But often when we increase the problem size, the ratio of communication to computation decreases, leading to improved efficiency.

Note that the parallel execution time T_p is at least: $T_p = \frac{T_s}{P} + T_{communication} = \frac{T_s}{P}(1+\gamma)$ (where γ is the communication to computation ratio).

For an algorithm to be scalable, the following must be true: Assume that the algorithm runs with efficiency η while simulating N atoms on P_1 processors. Then, given a larger number of processors P_2, one should be able to find a problem size N_2, such that the efficiency is η while simulating N_2 atoms on P_2 processors.

From the above equations, then:
$\frac{T_{s1}}{P1 \times T_{P1}} = \frac{T_{s1}}{P1 \times T_{P1}} \rightarrow \frac{T_{s1}}{P1 \times \frac{T_{s1}}{P1}(1+\gamma_1)} = \frac{T_{s2}}{P2 \times \frac{T_{s2}}{P2}(1+\gamma_2)} \rightarrow \frac{1}{(1+\gamma_1)} = \frac{1}{(1+\gamma_2)} \rightarrow \gamma_1 = \gamma_2.$

Thus, for efficiency to remain the same while increasing the number of processors, we must be able to find a larger problem size for which γ, the communication-to-computation ratio, remains the same. We next analyze several strategies for parallelizing molecular dynamics computations from this viewpoint of scalability.[1]

[1] In the following analyses, we focus on the volume of communication, i.e. the number of bytes of data sent or received by each processor. However, the cost of each message includes a relatively large startup cost on most current machines. This would argue in favor of counting the *number* of messages, rather than their volume. However, two

One of the earliest strategies used for parallelizing MD computations was to replicate all data (arrays containing attributes, coordinates and forces of atoms) on each processor. The force computations can then be evenly distributed across processors at will, as any processor is capable of carrying out any particular force computation. If there are N atoms, and P processors, the $O(N)$ forces accumulated by each processor must be added up across all processors. This requires communication time proportional to $N \log P$. Assuming cutoff, the amount of computation required is proportional to N (with a large proportionality constant), leading to N/P computation per processor. As the parallel execution time of the program is the sum of its computation and communication times, the parallel efficiency is affected by the communication time. However, the ratio of communication to computation times for replicated data (RD) is $P \log P$, and is independent of N. So, if we want to simulate a system twice as large as the current one, we cannot hope to double the number of processors to retain the same efficiency, because the fraction of time spent in communication will be larger. Thus, RD is said to be non-scalable. However, in practice, RD works effectively for tens of processors, and has the further advantage of being easier to implement, especially for an existing serial program. As a result, this strategy is used by many production quality MD programs, several of which are listed in Table 5.1.

A simple alternative to replicating data is atom decomposition (AD). The array containing atoms is arbitrarily partitioned across processors. Atoms that are close by in space need not be close by in the array. As a result, each processor may need data from all the other processors, leading to a $O(N)$ communication cost per processor, and loss of strict scalability.

Force decomposition (FD) is a technique that distributes the sparse force matrix in a blockwise fashion across processors. Each processor's share of the $N \times N$ force matrix is a block of size $(N/\sqrt{P}) \times (N/\sqrt{P})$. To compute this block the processor needs the coordinates of $2N/\sqrt{P}$ atoms, which come from $\sqrt{P}$ different processors. The communication cost per processor is thus $O(N/\sqrt{P})$ leading to the communication to computation ratio of $\sqrt{P}$. While this is much better than RD, it is still not scalable according to our strict criteria: as we increase the number of processors, the

factors allow us to focus on the volume. First, several message combining techniques exist that allow one to reduce the total number of messages. For example, if each processor is required to send a separate message to all other processors, this may require P messages in a straightforward implementation. But this can be reduced to $\sqrt{P}$ or even $logP$ messages, using techniques such as dimensional exchange. Secondly, in many of the algorithms analyzed here, (especially the spatial decomposition variants) the number of messages is a constant, thus obviating the need to consider it as a factor in asymptotic analysis.

Method	Communication cost per node	Theoretical scalability	Comment	Programs
RD	O($N \log P$)	Not scalable	Easy to parallelize	CHARMM [57] AMBER [58] UHGromos X-PLOR
AD	O(N)	Not scalable		EGO (Early version)
FD	O($N/\sqrt{P}$)	Not scalable	Tolerable communication costs	LAMMPS [59] CHARMM [60]
SD	$(N/P)^{2/3}$	isoefficiently scalable	Possible load imbalance	ddgmq [61] SIGMA
QSD	$O(N/P)$	isoefficiently scalable	Possible load imbalance	NAMD PMD [62] EulerGromos [63]
QSD+FD	$O(N/P)$	isoefficiently scalable	Load balance feasible	NAMD 2

Table 5.1 Scalability of decomposition strategies with examples of programs that use them. SD is spatial decomposition with exactly one domain per processor. QSD stands for *quantized* spatial decomposition, where the size of boxes is fixed, independent of the number of processors.

proportion of communication cost increases even if we simulate a larger biomolecule. Research by Plimpton [64, 59] and Hwang *et al* [60] shows that this method provides a better speedup than RD, and can be used with good speedups up to hundreds of processors. Results obtained by Hwang show that the proportion of communication cost goes from 6 percent of the computation cost on 32 processors to 36 percent on 128 processors in their particular implementation.

As the force matrix is sparse, and the atoms that are next to each other in the array are typically physically closer (note that the converse is false), one may get a non-uniform distribution of work with FD. Plimpton *et al* suggest randomly reordering the atoms to eliminate all traces of spatial locality, and restore load balance. (To handle symmetry arising out of Newton's third law, they compute the interaction between atoms i and j in one of the two possible blocks containing $F[i, j]$ and $F[j, i]$ respectively, depending on whether $i+j$ is even.) Hwang *et al* use a diametrically opposite technique: they use recursive bisection to partition the atoms so that nearby atoms are assigned to the same processor, and reorder the force matrix accordingly. The resultant load imbalance is handled by assigning an irregular-shaped piece of force matrix to each processor. The effect of this scheme has not been quantified formally. Empirically, the communication costs are seen to be smaller than AD.

The idea of assigning nearby atoms to the same processor, called spatial decomposition (SD), has been used and elaborated upon by several researchers. There are broadly three ways of doing this:

- Partitioning space into P boxes, one per processor.
- Partitioning space into fixed-size boxes, with dimension larger than the cutoff distance, requiring communication only between neighboring boxes.
- Partitioning space into a large number of small boxes, requiring each box to communicate with a large number of boxes.

With the first option, with sufficiently large N, the communication cost is proportional to the surface of the box, while the computation cost is proportional to its volume. Thus, this algorithm is highly scalable. However, the number of atoms is usually not as large as to validate this approximation. Also, this method is hard to use if the number of processors cannot be factored into three roughly equal factors.

With the second option, each box needs to communicate with a constant number (26) of neighboring boxes. If multiple boxes happen to be mapped to each processor, each will have to communicate with even fewer processors. Thus, the communication is proportional to N/P, and the algorithm is scalable with linear isoefficiency: If the program is running with a high parallel efficiency, and one doubles the size of molecule as well as the number of processors, one is assured that there will be the same number of boxes per processor and, therefore, the same parallel efficiency. This method was used in the earlier version of our program [65]. As other researchers have noted, load imbalance can be a severe problem for SD, especially for simulating non-periodic systems. NAMD addressed this problem by assigning the force computations corresponding to a pair of neighboring cubes (called patches in NAMD) to the patch on a processor with lower load. Although this strategy improves the load balance somewhat, it is clearly limited: neighbors of low-load patches (near the boundary) tend to be low-load as well. A second potential problem is that the number of processors one can utilize is limited to the number of patches.

The third strategy, implemented in EulerGromos by Clark *et al* [63], involves using boxes with sizes smaller than the cutoff distance, so that each box needs data from non-neighboring boxes. Again, multiple boxes can be mapped to each processor. The large number of messages resulting from the scheme can be reduced (but not the volume of data transferred) using a multi-stage communication scheme, first described in [66, 67] (called the "north–south–east–west" exchange algorithm; also called the "shift algorithm" in [63]). A prototype implementation of EulerGromos has demonstrated good speedups with the strategy. The programs UHGromos (with replicated data) and EulerGromos were written by the same team, and a performance comparison showed that EulerGromos was faster if there are enough processors, the crossover point in their experiment being 64 processors [63]. In our own experimentation, we found this "small boxes" strategy to lead to high overhead, possibly due to the following reason: when multiple time stepping is used, the cutoff distances are short (e.g. 8-9 Å). As a result, boxes with the width half of the cutoff distance are too small (with about 12 atoms per box) to amortize the overhead.

In another variant of spatial decomposition, the FAMUSAMM algorithm implemented in latest versions of EGO [68] uses hierarchical decomposition of space, based on structural features of biomolecules, to implement a structure-adapted multipole algorithm.

The strategy used in NAMD 2 can be thought of as a synthesis of space decomposition and force decomposition. NAMD 2 decomposes atoms into boxes of size $\mathit{cutoff} + \mathit{margin}$, and distributes these boxes to processors as described above. (The margin parameter will be explained in Section 5.3.1.) In addition, NAMD 2 allows the computation of nonbonded forces between each pair of neighboring boxes to be freely assigned to any processor. The number of such pairs is 13 times the number of boxes. Thus, in principle, NAMD 2 can utilize a much larger number of processors than a purely spatially decomposed program (such as the original NAMD). However, to realize the potential efficiency, NAMD 2 requires sophisticated load balancing. Measurement-based object migration techniques are ideally suited for providing such load balancing. This load balancing strategy, and its object oriented implementation, is elaborated in Section 5.3.

5.2.1 Parallelization of Bonded Force Components

Parallelizing computation of bonded forces in the context of spatial decomposition is also challenging. Bonded forces represent covalent chemical bonding, and are parameterized on distances,

PE 0 Patch (0,0)	PE 1 Patch (0,1)	PE 2 Patch (0,2)
PE 3 Patch (1,0)	PE 4 Patch (1,1)	PE 5 Patch (1,2)
PE 6 Patch (2,0)	PE 7 Patch (2,1)	PE 8 Patch (2,2)

Figure 5.1 Neighboring patches in the two-dimensional downstream patch scheme.

angles between atom triplets (angles), and angles between planes formed by four atoms (dihedrals and impropers). Here we will discuss the parallelization of angles.

We can assume that the three atoms involved in an angle are within a $2 \times 2 \times 2$ box of neighboring patches. This is valid as long as bond lengths are smaller than half of the patch dimension. Assume that each patch is on a separate processor. The simple implementation (used in the original version of NAMD) is to have each patch send atom coordinates to each of the 26 neighboring patches. Each patch looks at the neighboring coordinates and computes the forces on its atoms due to the angles associated with the neighboring patch data. In this scheme these computations are duplicated for those angles involving atoms in multiple patches.

A simple reorganization of the computation allows one to reduce the number of processors to which each patch must deliver position data. Figure 5.1 illustrates the algorithm in a two-dimensional decomposition. The older algorithm would require patch $(1,1)$ to send eight position messages. The new algorithm uses the rule that the angle forces among three atoms in patches (x_1, y_1, z_1), (x_2, y_2, z_2), and (x_3, y_3, z_3) are computed on the processor owning patch $(\min(x_1, x_2, x_3), \min(y_1, y_2, y_3), \min(z_1, z_2, z_3))$. Note that this may be a processor where none of the three atoms reside. A patch (a_1, a_2, a_3) sends its data only to *downstream patches* with coordinates (b_1, b_2, b_3) such that $a_i - 1 \leq b_i \leq a_i$. In the two-dimensional case of figure 5.1, patch $(1,1)$ needs to send data to only three other processors, $(1,0)$, $(0,1)$, and $(0,0)$. So now an angle composed of atoms in patches $(1,1)$, $(2,1)$ and $(2,0)$ is calculated on the processor that owns patch $(1,0)$. In three dimensions, there are seven downstream patches with whom each patch communicates (instead of 26). In addition, this algorithm avoids redundant computation by applying Newton's third law, although this increases the number of messages to 14 per patch to return the computed forces. Even when there are multiple patches per processor, the new scheme still results in fewer messages than the straightforward implementation. This method has similarities to work done by Brown *et al* [69], except that Brown's method requires additional communication, including a global reduction to guarantee that each angle is computed on exactly one processor. They also use a staged communication strategy, also described by Clark [63]. This strategy reduces the number of messages from seven to three by combining messages, but the *amount* of data exchanged remains the same. Esselink and Hilbers [70] consider a similar scheme, but opt in favor of one requiring communication from half of the neighbors within the cutoff distance, citing the complexity involved in identifying the angle forces to be calculated on a processor. Our scheme, along with the tuple search algorithm described in Section 5.3.2, demonstrates that an efficient implementation is possible.

5.3 Design

The three overall goals of NAMD 2 were improved sequential performance, scalability with respect to both simulation size and number of processors, and modifiability. We also wanted the program to deal with periodic as well as non-periodic configurations, and provide the option of calculating long-range electrostatic forces. Modifiability was required to support ease of maintenance as well as the ability to add new features. Each of the design decisions described below contributes to achieving these goals.

The hybrid spatial/force decomposition scheme described above provides the basis for scalability of NAMD. This scheme requires that a number of entities (multiple "patches" and force computations) reside on each processor. Instead of a monolithic program that orchestrates all these diverse actions on a single processor, we chose a message-driven object based paradigm. Each of the computational entities were implemented as objects, and they are scheduled based on availability of data needed for their continued execution. With this, a force computation (for example) is scheduled for execution only when all the data it needs is available on the local processor, thus avoiding the possibility that any particular entity will block the processor while waiting for specific data.

Like the original version of NAMD, NAMD 2 uses CHARMm force fields and X-PLOR coordinate and molecular structure files. In addition to non-periodic simulations, NAMD 2 can use periodic boundary conditions over any combination of the three coordinate axes. It performs cutoff simulations or full-electrostatic simulations employing multiple timestepping using the DPMTA [71] or DPME [72] libraries. NAMD 2 can connect to VMD [73], the visualization component of MD-SCOPE [74], to allow monitoring of and interaction with ongoing simulations.

We will first describe the core structure of NAMD 2. The specific algorithms and interesting data structures used for force computations are described next. To make a modular implementation of NAMD 2 possible, it was necessary to utilize and integrate several different parallel programming paradigms. How such multiparadigm programming was supported and why it is useful is discussed in section 5.3.3.

5.3.1 Object-Oriented Design

For the sake of extensibility, we designed a class hierarchy that incorporates the structure of molecular dynamics. The core of this hierarchy is constituted of *Patch*, *Compute* and *Sequencer* objects. Patches represent cubical (actually, orthorhombic) regions of space, and all the atoms within such a region. *Compute* objects signify computation of different kinds of forces on atoms in a set of patches. A *Sequencer* specifies the numerical integration scheme, and drives the simulation of atoms within one patch. These base classes, and associated service code, define basic functionality needed in any MD computation. A programmer wishing to add features to NAMD 2 can often overload an existing class to add some particular functionality.

Patches

The simulation space is divided into regular cubic regions called *patches*, each of which is responsible for atoms within its boundaries, their coordinates, and the forces exerted on them. The dimensions of the patches are chosen to be slightly larger than the cutoff radius. Thus one can make the assumption that each patch only needs the coordinates from the 26 neighboring patches to compute non-bonded forces. To ensure that all atoms in a patch indeed belong there, it is necessary to check all atoms at the end of each timestep, and migrate them to their proper home patches, if needed. This will be prohibitively expensive. Using a well-chosen margin allows one to optimize this: With margins, even if an atom strays outside the coordinate space of its patch, the assumption stays valid as long as the straying is less than the margin. NAMD 2 uses 4-8 timesteps before migrating atoms to new locations, and detects if there are any violations of the above assumption. The margin is normally chosen to be about 1.5Å.

Compute Objects

The compute objects, together, are responsible for all the force computations in NAMD 2. Each compute object is associated with 1, 2 or more patches. The compute object utilizes coordinates and other information from these patches, calculates a specific type of force (e.g. nonbonded), and provides its client patches with the forces it calculated. The runtime code manages dependencies: once a compute object has informed the system of the patches it depends on, the system triggers each compute object when its client patches are ready with the coordinates for the next time step. After the forces have been calculated, the compute object informs the patch so that it can take the next step in its algorithm. Adding a new type of force computation thus becomes a simple matter of specifying which patches a compute depends on, and providing the code to calculate the new force terms.

Proxies

If a compute object and the patches it depends on reside on different nodes, then the compute object must get the coordinate information from the patches on remote processors, and send forces back to them. Often, there will be multiple compute objects on a processor that are dependent on data from the same remote patch. To avoid redundant communication in this common case, a representative of the home patch called a *proxy patch* is created on the processor. The compute object then contacts the proxy patch for the atom positions, and deposits forces into the local proxy as well. Thus each patch sends all its coordinates to its proxies, and receives forces from them once per time step. As the average number of proxies is often substantially smaller than the number of compute objects that depend on a patch, this represents a considerable optimization. In addition, the proxy patch provides the compute objects with the same interface as the home patch, enhancing the ease of programming: except for the code that deals with updating coordinate

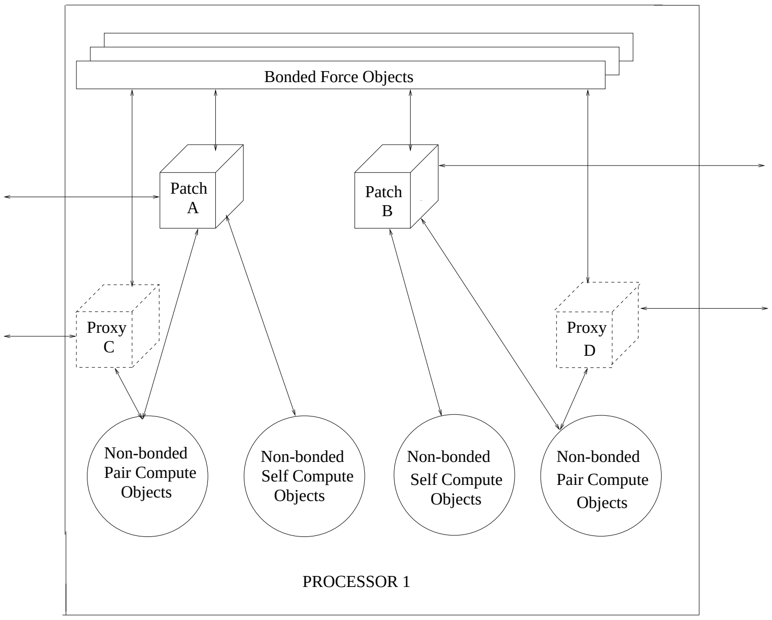

Figure 5.2 In NAMD 2 forces are calculated by independent compute objects which depend on one or more patches for atomic coordinates. Proxy patches (shown as dotted cubes) are used to reduce the communication cost. Communication between the different entities are represented by arrows.

information in proxies at the beginning of each timestep, and sending forces back at the end, the rest of the program can be written without any communication code. (Some communication is needed to form global sums of various energy terms, but again each object simply deposits its contribution into an existing library which does the necessary communication). Figure 5.2 shows how compute objects interact with patches and proxies.

Sequencers

Every home patch has a *sequencer thread* associated with it. It loops over all the timesteps in the simulation and essentially describes the life cycle of a single patch. The sequencer provides the strategy, while the other objects such as the compute objects and patches provide the mechanisms. The *sequencer* abstraction allows it to be replaced by alternate modules for experimenting with different algorithmic strategies, thus enhancing modifiability, one of the primary design goals of NAMD 2.

The pseudo code for the sequencer algorithm is shown in figure 5.3. The algorithm uses multiple time stepping to reduce the computational cost of computing electrostatic forces due to all pairs

```
{
    runComputeObjects();
    addForceToMomentum(0.);
    submitReductions(step);
    submitCollections(step);
    rescaleVelocities(step);
    berendsenPressure(step);
    langevinVelocities(step);

    for ( ++step; step <= numberOfSteps; ++step )
    {
        addForceToMomentum(0.5*timestep);
        if (doNonbonded)
                addForceToMomentum(0.5*nbondstep,Results::nbond);
        if (doFullElectrostatics)
                addForceToMomentum(0.5*slowstep,Results::slow);

        addVelocityToPosition(timestep);

        runComputeObjects(!(step%stepsPerCycle));

        addForceToMomentum(0.5*timestep);

        if (doNonbonded)
                addForceToMomentum(0.5*nbondstep,Results::nbond);
        if (doFullElectrostatics)
                addForceToMomentum(0.5*slowstep,Results::slow);

        submitReductions(step);
        submitCollections(step);
        rescaleVelocities(step);
        berendsenPressure(step);
        langevinVelocities(step);
        rebalanceLoad(step);
    }

    terminate();
}
```

Figure 5.3 Sequencer algorithm for a dual-timestep long-range non-bonded force simulation of particles in the system. The total electrostatic force acting on each atom is divided into two parts, a short-range (local) component consisting of forces due to pairs that are separated by less than the local interaction length (electrostatic cutoff) and a long range component consisting of the remaining pairs. The long-range forces vary slowly, hence they are evaluated less frequently.

In each timestep, depending on the frequency parameters specified by the user, it either computes

1. only the bonded forces, or
2. bonded as well as non-bonded forces within a cutoff, or
3. bonded as well as non-bonded forces

The velocities are updated based on the forces, velocities in turn are used to update the positions. The sequencer then informs the home patch that its positions are ready. At every cycle boundary, the patch sends the atom positions that don't belong to its space to the neighboring patches. The proxy patches are populated next. The home patch sends all its atom positions to all its proxies. Once the proxies are populated, it notifies all the dependent compute objects that the coordinates are available for computing the forces for the next timestep and suspends the sequencer thread. The compute objects calculate the forces and awakens the sequencer. The newly calculated forces are used to update the velocities, and reductions and collections are initiated for the calculation of total energy and trajectory output.

5.3.2 Force Computation Schemes and Algorithms

The specific algorithms used to carry out and, in the case of bonded forces, partitioning the computation, are influenced by the parallel decomposition scheme. The methods discussed below contribute significantly to both the serial and parallel performance of NAMD 2, as well as to its modifiability and clarity of design.

Non-bonded Interactions

The calculation of local (cutoff) electrostatic and Lennard-Jones components of the force field is the most expensive step in a simulation. As such, it must be calculated both efficiently and in a manner which can be readily parallelized and load-balanced. We have enabled parallelization within the NAMD 2 framework by dividing most of the non-bonded calculation between two classes of compute objects: self and pair. The self compute objects calculate pairwise interactions between atoms residing within a particular patch. The pair compute objects calculate pairwise interactions between pairs of atoms residing in neighboring patches. The size of a patch is guaranteed to be sufficiently large that all atoms not in neighboring patches are beyond the cutoff distance. A third class, the exclusion compute objects, efficiently handles the modifications to the non-bonded force field related to the connectivity of the molecule.

Bonded Interactions

A trivial operation in a serial program, the efficient calculation of the bond, angle, dihedral, and improper terms of the force field is complicated by the separation of atoms into patches. The atoms required for a single dihedral calculation may reside on up to four patches. The calculation of these bonding terms must be divided among compute objects on different nodes without introducing excessive communication or overhead.

In NAMD 2, a single compute object for each type of bonded interaction (bond, angle, dihedral, improper, and exclusion) is created on each node. This compute depends on all patches local to that node and the proxies of non-local patches which are *upstream neighbors* of local patches, defined as being adjacent to a local patch while having greater or identical x,y, and z coordinates. The proxies of upstream patches are also sufficient to calculate non-bonded interactions, so no extra communication is necessary (although migrating compute objects to improve load balance will require additional proxies). All of the coordinates for a particular bonded interaction are guaranteed to be present on the node containing their *common downstream patch*, the patch with the minimum x, y, and z coordinates of the patches containing the atoms of that interaction.

Developing a good algorithm for dividing the bonded calculation among the processors turned out to be much more important than anticipated. The original design called for every patch to have access to all of its neighboring patches. Bonded forces would be calculated for all patches on a node, but no forces would be returned to proxies. This eliminates the need for a point of synchronization and allow proxies to return results as soon as possible. The current algorithm was devised to more than halve communication when running on large numbers of nodes while eliminating redundant force calculations.

5.3.3 Multiparadigm Programming Approach

One of our major design decisions to use the multiparadigm approach for implementation of NAMD 2. A multiparadigm approach restricts the kinds of load balancing strategies which can be used, since those strategies must understand the issues affecting the load of a variety of components implemented using different parallel-programming paradigms. Measurement-based strategies are one category capable of handling such diverse sets of components. The multiparadigm implementation of NAMD 2 was based on the following requirements:

- NAMD 2 must be able to utilize the DPMTA module which is written in PVM [20], without having to pay the overhead of interacting with separate PVM processes.
- The performance-critical core of parallel computation must be written in Charm++ [5], to take advantage of its message-driven execution for adaptive overlap of communication with computation, and the ability to prioritize messages.
- The coordination logic of the program must be modularly separated from the parts of the program that implement the parallel mechanics needed to execute it.
- Nearly half of the code involved parallel initialization, using synchronous message passing. Reusing code from the original NAMD without extensive modification was desirable.

Traditional implementations of these paradigms cannot work together in a single application, because the control-regimes for these paradigms are seemingly incompatible. Fortunately, Converse [4], described in the next subsection, allowed us to use these different paradigms in NAMD 2.

Converse: An Interoperable Run-time System

Converse provides the portable framework that supports multilingual parallel programming by using (a) a common scheduler's queue for entities across languages, (b) by providing a customizable scheduler that is exposed to language run-time systems, and (c) by providing a component-based architecture with support for many common building blocks of parallel run-time systems.

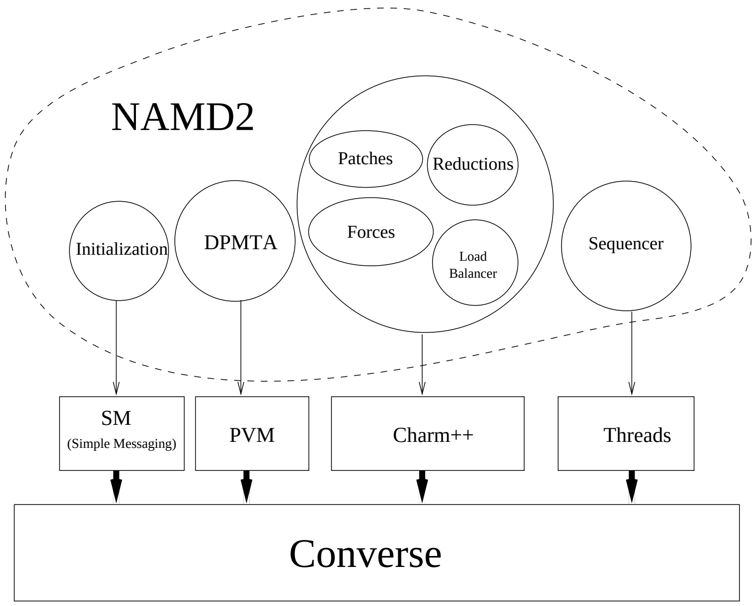

Figure 5.4 NAMD Architecture

A specific language implementation in the Converse framework incorporates only the necessary components, customizing them as needed, thus incurring minimal overhead. The components provided by Converse include a machine interface (that supports communication, timers and other operating system calls), scheduler queues, a threads package, and a message manager library. Over a dozen languages based on different paradigms, including Charm++ and PVM, have been implemented on top of Converse [75].

Converse has been implemented on several parallel platforms, including workstation cluster, IBM SP2, Convex Exemplar, SGI Origin 2000, Cray T3E, and Intel Paragon. Porting Converse to a new machine is relatively easy because only a small machine-dependent component needs to be rewritten.

Paradigms Used in NAMD 2

Using different languages for implementing different modules, we were able to address all of the above requirements (see Figure 5.4.)

The initialization code involved reading in the data from the disk, and installing individual data structures on the parallel processors. To avoid race conditions, it used synchronous message passing, with tagged messages and blocking receives. SM language provides such message passing with lower overhead than PVM or MPI. Also, in SM, an innovative use of message streams allowed us to streamline messages without fully assembling them. This resulted in major saving in memory requirements in the initialization phase.

We were able to reuse the DPMTA module by simply linking it with the implementation of PVM on Converse.

The remaining requirements presented a major challenge. Writing the code for core computations entirely in Charm++ would lead to good performance, and also reasonable perspicuity. However, we wanted to separate the algorithmic logic from the parallel logic, so that someone wishing to change the basic numerical algorithm (e.g. Verlet-I integration, Langevin dynamics) can do so without being encumbered by the parallel logic. This was accomplished by observing that the algorithmic logic involves the description of the life cycle of a patch: sequencing of the steps of the algorithm, and choosing individual sub-algorithms for each step. In contrast, the parallel logic involves movement of data, triggering of computations based on the availability of data, and interprocessor synchronization. The latter can be thought of as mechanisms under control of the algorithmic logic, encapsulated in the sequencer object.

As there are multiple patches per processor, multiple sequencers must be active on individual processors, with the possibility that they may not be in lock-step. Some sequencers may be lagging behind, while others race ahead, constrained only by the data dependencies. Periodically each sequencer must wait for its data to become available, while one of the other sequencers on the processor can continue. Also, the bulk of parallel computations in Charm++ modules must be allowed to execute interleaved with the sequencers. This suggested the use of threads for implementing sequencers. Normally, it would be difficult to concurrently run multiple threads and message-driven objects in the same program. For one thing, Charm++ would use its own message-driven loop to schedule messages for its objects, which would conflict with the thread scheduler. Running Charm++ in one of the threads is not a good solution, because of the stack space limitations imposed by threads, and the context switching overheads of threads which we wish to avoid for the fine-grained dominant computations. However, the threads implemented within Converse allow applications to mix threaded and non-threaded computations effectively. The common scheduler can switch between threads and objects, while allowing unlimited stack space to the objects.

In spite of the modular separation of the sequencing operations provided by the use of threads, two significant problems remained in using threads. The first problem was the overhead of thread context-switching, and the second was wastage of memory space due to over-estimating the stack size needed for threads. Our experiments comparing the overheads of thread context switching to the message-driven scheduling showed [76] that in many programs, this overhead can be a significant factor in program performance. Also, our efforts to run NAMD 2 on machines with limited swap space indicated that many real parallel machines do not have enough memory to run thousands of threads. However, creating thousands of message-driven objects is almost always possible on these machines. Our research on creating a low-overhead and lightweight programming abstraction resulted in Structured Dagger, a coordination language for message-driven parallel programming [76], implemented on top of Charm++. We have since re-implemented one of the sequencer algorithms using this language.

5.4 Scalability and Parallel Performance

The previous sections have described how NAMD 2 was designed to efficiently partition the MD problem into a large number of components which can be executed in parallel for good scalability. However, that scalability is not realized unless those components can be distributed properly on the parallel machine. Since those components implement many different algorithms with various data dependencies and computational requirements, the measurement-based approach we have developed is especially well-suited to this problem. This section describes the load balancing algorithm used by NAMD 2 and the resulting performance.

5.4.1 Load Balancing

The parallel efficiency of NAMD 2 is highly dependent on efficient load distribution. Computational time is consumed mainly by two object types, patches and force objects. To distribute these objects, NAMD 2 performs two types of load balancing, initial load balancing and dynamic load balancing.

Initial load balancing occurs at program startup. Patches are distributed using a recursive bisection algorithm, to distribute atoms approximately evenly, while still maintaining data locality.

All of the bond force objects are distributed next, one per processor. Finally, the non-bonded force objects are distributed. Non-bonded force objects for patch self interactions are first placed on the same processors as their patches. Patch-pair force objects are next distributed to the *upstream processor* for the two patches. This guarantees that communication is initially minimized, since the same proxies required for the bond computations are also used for the non-bonded force objects. It does not necessarily result in an optimum load balance, since patch locality probably results in clusters of computationally intensive force objects being distributed to a small number of processors, but by minimizing communication, the strategy promotes effective dynamic load balancing.

NAMD 2 relies on dynamic, measurement-based load balancing scheme to achieve efficient throughput. Heuristic-based static load balancing schemes exhibit several disadvantages in molecular dynamics. The first difficulty we ran across was determining the computational load represented by a particular force object. Processor speed and inter-processor communication costs both affect the heuristic model of load necessary for static load balancing. Determining an accurate model for all the different architectures NAMD 2 runs on was prohibitively time consuming. Therefore, NAMD 2 was designed to measure the actual time consumed by various objects and use that data to redistribute these objects efficiently. This method produces the additional benefit that it can be repeated during a run, to react to changes in load distribution as the atoms move during the evolution of the simulation.

Migratable objects are those objects which can be moved to a new processor during a simulation. Ideally, any work should be assignable to any processor, but making objects migratable complicates their code. In NAMD 2 we chose to make only non-bonded compute objects migratable. These objects consume the majority of the simulation time, yet there are enough of them to provide sufficient freedom to balance the load. The load balancer treats non-migratable work, including integration and bond force evaluation, as background load which does not change, but must be considered when distributing the migratable objects.

Simulations execute the following procedure to perform load balancing. First, the simulation runs for a small number of steps, typically lasting a few minutes. Migratable object times are measured during this time. In addition, the Converse system provides a means of measuring idle time for each processor, so non-migratable load is determined by $t_{non\text{-}migratable} = t_{total} - t_{idle} -$

$t_{migratable}$. After a particular simulation step, the program collects the load data from each processor, computes a new load distribution, and moves the migratable objects.

The load balancing algorithm we selected uses a modified greedy algorithm to distribute load while keeping the amount of communication small. A min-heap stores the currently-assigned load on each processor, so the most lightly-loaded processor can be easily retrieved. The initial processor load is the measured background load for that processor. Migratable objects are also stored in a max-heap to allow the most expensive compute objects to be assigned first. The load balancer also uses a map of required proxies, ie., the proxies required for the calculations of bonded forces. The load balancer makes no distinction between patches and proxies, so when the following description mentions proxies, it also refers to patches. An average load is computed from the total measured load. Then each migratable object is examined, starting with the most expensive. The load balancer considers three possible locations for the compute: the least-loaded processor already receiving both proxies, one proxy, or no proxies. The object is assigned to the two-proxy processor if the resulting load on the processor is less than the average load times an overload factor. Otherwise the one-proxy processor is considered, and finally, if necessary, the no-proxy processor. Then the load balancer adds the object load to the selected processor's total load, and updates the proxy map. A fairly generous overload factor is selected, so that data locality is more important than exact load balance. This procedure repeats until all compute objects are assigned.

A refinement procedure is next used, to fine-tune the load balancing produced by the sorting algorithm. The load balancer makes a second pass through the processor assignments. A smaller overload factor is selected, and any processors whose load exceeds the average by more than the overload factor has a number of force objects moved to lighter loaded processors, using the same selection criteria described above. Since the threshold is now lower, this procedure is likely to add a few more proxies, but produces good load balance with small communication overhead.

After refinement, the simulation resumes using the new object placements. Although the new load balance is good, the changes in the communication patterns produce changes in the background load, which the previous load balancing did not take into account. Therefore, the simulation runs for a few more steps, then stops for a second load balancing. This time the load balancer only performs the refinement step, resulting in just a few adjustments to the object placement. The

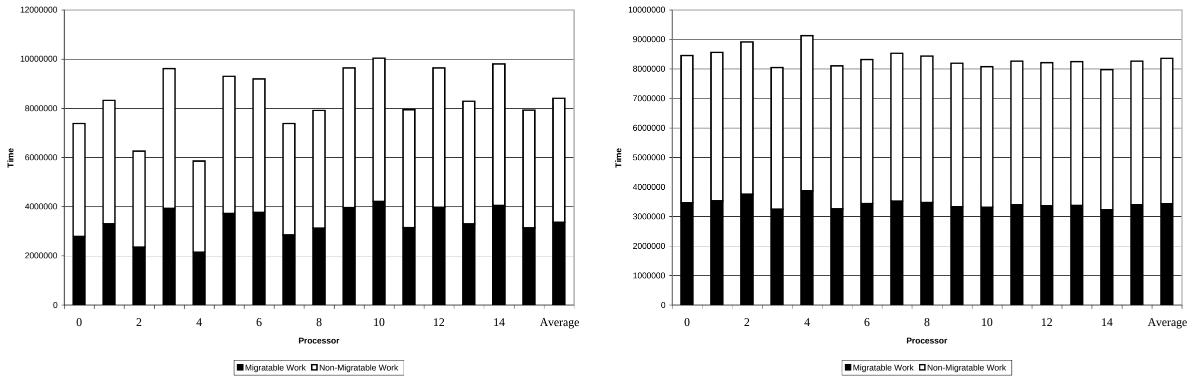

Figure 5.5 Load distribution for a 16 processor simulation, showing the load before (left) and after (right) running the load balancer.

resulting load assignment produces well-balanced load distribution, as shown in Figure 5.5.

5.4.2 Parallel Performance

NAMD 2 has demonstrated good performance on many parallel machines. Table 5.2 shows the performance data obtained while running three benchmarks of varying sizes on the Cray T3E. The systems chosen range in size from 3,800 atoms to 92,000 atoms. The largest configuration used (ApoA-I) was a periodic simulation, while the other two were non-periodic. The tables show time per integration step in seconds, and the corresponding speed-up obtained. In some cases, the memory per processor was insufficient to allow runs on small numbers of processors. In those cases, the speed-up has been estimated using the speed-up factors obtained on smaller molecules. This is conservative, as the speed-ups typically get better when a larger number of atoms are involved on the same number of processors. Figure 5.6 demonstrates that NAMD 2 achieves good scaling even on large numbers of processors, for sufficiently large simulations. As can be seen, speed-ups around 100 were obtained with 128 processors, and they seem to increase with the same efficiency beyond 128 processors. The speed-ups were calculated in comparison with the one processor performance of NAMD 2.

NAMD 2 is designed to be portable. Porting NAMD 2 to new machines requires us to modify the program to satisfy the vagaries of C++ compilers on different machines. Other than that, porting is straightforward. In particular, no machine specific performance tuning is necessary. Table 5.3 shows the performance obtained while simulating ER-ERE on a variety of parallel machines.

Simulation # of atoms		Processors								
		1	2	4	8	16	32	64	128	160
bR	Time	1.138	0.578	0.315	0.158	0.086	0.048			
3,762 atoms	Speedup	1.0	1.97	3.61	7.20	13.2	23.7			
ER-ERE	Time		6.115	3.099	1.598	0.810	0.397	0.212	0.123	0.098
36,573 atoms	Speedup		(1.97)	3.89	7.54	14.9	30.3	56.8	97.9	123
ApoA-I	Time			10.760	5.464	2.850	1.470	0.729	0.382	0.321
92,224 atoms	Speedup			(3.88)	7.64	14.7	28.4	57.3	109	130

Table 5.2 Execution time (seconds) per timestep for several simulations on the CRAY T3E (450MHz processors). All the simulations were run using a 12Å cutoff. Some simulations could not run on small numbers of processors due to lack of memory, so speed-up numbers in parentheses are estimates.

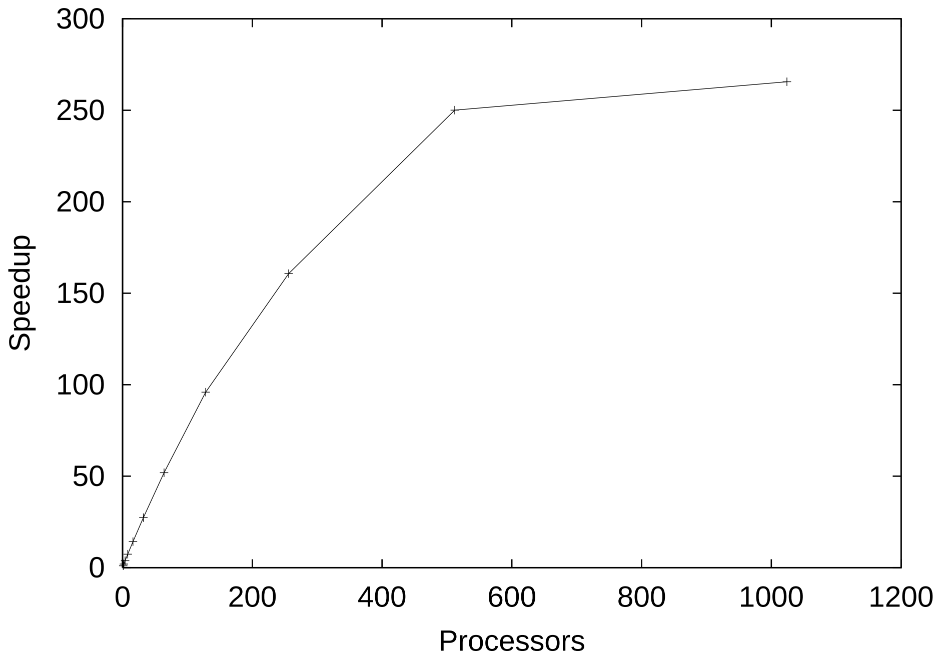

Figure 5.6 The speed-up for ApoA-I (92,224 atoms, 12Å cutoff) on the ASCI-RED.

		Processors								
		1	2	4	8	16	32	64	128	160
T3E 450MHz	Time		6.12	3.10	1.60	0.810	0.397	0.212	0.123	0.098
	Speedup		(1.97)	3.89	7.54	14.9	30.3	56.8	97.9	123
Origin2000 195MHz	Time	8.28	4.20	2.17	1.07	0.542	0.271	0.152		
	Speedup	1.0	1.96	3.80	7.74	15.3	30.5	54.3		
ASCI-RED 200MHz PPro	Time	28.0	13.9	7.24	3.76	1.91	1.01	0.500	0.279	0.227
	Speedup	1.0	2.01	3.87	7.45	14.7	27.9	56.0	100	123
HP735/125 125MHz	Time	24.1	12.4	6.39	3.69					
	Speedup	1.0	1.94	3.77	6.54					
LINUX 400MHz PII	Time	12.34	6.42	3.28	1.69	1.01	0.62			
	Speedup	1.0	1.92	3.77	7.30	12.21	19.90			

Table 5.3 Execution time (seconds) per timestep for ER-ERE (36,573 atoms, 12Å cutoff) on several parallel machines. The HP735 workstations are interconnected with a 100Mb/s ATM switch. The Linux machines use 100Mb/s switched Ethernet.

Although the parallel performance of NAMD 2 is quite good, it still leaves room for improvement. The efficiency on 128 processors is close to 80 percent. Although part of the efficiency loss is inevitable due to communication overhead, we believe that further improvement is possible. Our analysis indicates that about half of this loss (ten percent) is due to communication overhead and increased time for some of the force calculations when done in parallel. The other ten percent is idle time, spent waiting for messages. This is due to a combination of load imbalance and message scheduling. We believe that improvements to our load balancer, combined with the use of priorities for messages (which are supported by Charm++) will allow us to decrease the idle time further.

Some of the features of C++ do impose a speed penalty, so profiling information was employed to pinpoint which parts of the code penalized performance, and these portions were hand-tuned for best performance.

NAMD 2 currently does not use pair-lists, relying instead on the fact that only a small subset of the atoms (those in the neighboring patches) must be examined for each atom. Although this strategy yields good performance, it is possible that either incorporating pair lists, or partitioning atoms into cells smaller than the patch-size will yield further performance improvements.

The current load balancer does not take the interconnection topology of the host computer into account. This is valid for many parallel machines, such as the Cray T3E. However, an increasing number of parallel machines exhibit a two-level architecture: a collection (from two to thirty-two) of shared memory processors, connected to other similar nodes via an interconnection topology. In other machines, the communication time (and bandwidth utilized) varies widely depending on which processors are communicating. We plan to develop a load balancing strategy that adapts itself to the communication architecture.

Our benchmarks above describe only cut-off electrostatics performance of NAMD 2. The long-range interactions in NAMD can be computed using the DPMTA library developed at Duke University. The parallel performance of DPMTA has been documented in [71]. The performance of the long-range interactions phase is orthogonal to that of the rest of NAMD 2. In the future, we plan to experiment with other methods for long-range electrostatics, such as particle-particle particle-mesh methods, and with tighter integration of such libraries to improve performance.

5.5 Porting NAMD 2 to the Load Balancing Framework

Since the techniques used for load balancing in NAMD are similar to the techniques generalized and embodied in the load balancing framework, it was conceptually simple to port NAMD to the load balancing framework. However, in an application as complex as NAMD, there are many implementation detail which complicated the porting procedure.

NAMD does not use object arrays, so it implements its own procedures for migrating work. The migratable work in NAMD is composed of compute objects, which were originally timed and migrated by NAMD itself. The most important modification to NAMD was to move these functions to the framework. NAMD originally used a `LdbCoordinator` object to perform these functions. This object was rewritten to instead serve as the interface between NAMD and the framework. At creation, the `LdbCoordinator` registers itself as an object manager on each processor, and registers each compute object with the framework. Every time a compute object executes, the `LdbCoordinator` informs the framework of the start and end of the computation. When the framework requests that some objects migrate, the `LdbCoordinator` records the migration, makes the registration calls required by the framework, and performs the multi-step process required by NAMD to actually move the objects to its new location.

NAMD does not currently report communication to the framework. The difficulty with using the framework to record communication involves the optimized proxy communication described in

Section 5.3. There are two classes of performance-critical communication in NAMD. First, patches multicast their atom positions to a group of associated compute objects. Then, these compute objects perform a vector reduction to sum their calculated forces, providing a total force for each atom. In NAMD this is implemented using proxies to reduce the total message count, to eliminate sending the same data more than once for compute objects on the same processor, and to parallelize the force summation. The naive way to record this communication with the framework would be simply to register each proxy, and to inform the framework about each proxy communication. However, this is not useful because a particular proxy's future communication pattern is determined by the location of compute objects after migration, not their prior distribution.

We have devised two ways of extending the framework to correctly record the proxy communications used in NAMD. Both methods require adding new types of communication in addition to the types already present (object to object, processor to object, and object to processor). One was is to recognize that the proxy communication pattern is actually a combination of a multicast from the patch to the destination computes, followed by a reduction from those computes to the originating patch. The NAMD proxy object is merely the object that implements these operations efficiently. Both multicasts and reductions among groups of objects are general communication primitives that occur in many applications, so writing load balancers to understand these primitives is a useful extension to the framework. A second way to record proxy communication is simply to add a proxy communication type to the framework. Although not as general as the first method, it is still useful across a variety of applications, and provides additional information (the linkage between the multicast and the reduction) which may be useful to some load balancing algorithms.

Since these communication types are not currently part of the framework, NAMD does not record communication with the framework at all. This means that, although general-purpose load balancers function with NAMD, they do not do well at minimizing communication costs. Instead, the existing NAMD load balancer has also been modified to use the framework for object timing, but to use the object-mapping information stored in various data structures within NAMD to determine communication patterns.

The production NAMD code now includes the framework-based load balancing code, and the most recent performance optimizations were implemented in the framework-based code. Table 5.4

Processors	Time (s/step)	Speedup	GFLOPS
1	57.1	1	0.0481
4	14.7	3.9	0.186
8	7.31	7.8	0.375
32	1.89	30.1	1.45
64	1.04	55.1	2.63
128	0.504	113	5.43
256	0.253	225	10.8
384	0.181	315	15.1
512	0.146	390	18.8
768	0.133	429	20.6
1024	0.0861	663	31.8
1536	0.0667	855	41.1

Table 5.4 Latest program performance for the ApoA-I simulation ASCI Red

shows the most recent performance for ApoA-I on the ASCI-Red. The speedup of 855 is the highest reported for a cutoff simulation of a biomolecular system.

5.6 Lessons from NAMD

We have described the design of the parallel molecular dynamics program NAMD 2. The program uses a novel combination of spatial and force decomposition to attain both theoretical and practical scalability. It uses an adaptive load balancing scheme to attain high parallel efficiency even while simulating non-periodic systems. It uses a highly modular design with a class hierarchy implemented in C++ that incorporates mechanisms needed in molecular dynamics codes while permitting easy experimentation with algorithmic strategies. Using Charm++ allows adaptive overlap of communication and computation, and using the Converse interoperable run time system allows NAMD 2 to integrate modules written in different parallel paradigms. Performance results obtained on a variety of parallel machines demonstrate the speed and parallel efficiency of NAMD 2.

Molecular dynamics simulation presents complex load balancing challenges. The pieces of work which execute in parallel represent widely varied amounts of computation, and the algorithm exhibits an appreciable amount of communication. We found it difficult to devise good heuristics for a static load balancer, and therefore used measurement-based dynamic load balancing that was accurate and simpler to implement, especially when object migration is supported by the programming paradigm.

NAMD demonstrates that measurement-based load balancing and object migration works well in real, production-quality applications. NAMD has also demonstrated that, for some applications, load balancing strategies must take communication as well as computational load into account. Experience with NAMD load balancers has contributed to the the design of several of the framework load balancers described in Chapter 6.

NAMD 2 also demonstrates that although many of the load balancing mechanisms we implemented could have been implemented simply using the load balancing framework, the application design still must provide sufficient parallelism to achieve efficiency on large parallel machines, often a complex design problem. In NAMD 2, the decomposition of the computation into a large number of migratable objects provides the sufficiently-fine granularity for good parallel performance. Other applications must also be designed with enough parallelism to give the framework sufficient control over the distribution of the application load. Nevertheless, the framework eliminates the burdensome tasks of implementing load balancing mechanisms and strategies.

CHAPTER 6

LOAD BALANCING STRATEGIES

Our work on automatic load balancers has proceeded along two axes. First, we have devised and advanced application-specific load balancer for a production-quality molecular dynamics application, as described in Chapter 5. Second, we have developed several load balancing strategies within the load balancing framework, and evaluated them with a parallel load-balance test program, `lb_test`, and several applications. Our intention in not to show the superiority of the strategies implemented; rather, we are demonstrating how the framework allows us to create a variety of effective, general-purpose strategies.

6.1 Load Balancing Strategies

The fundamental tradeoff in parallel load balancing is the balance between execution time and load balancing time. Load balancing algorithms may be able to improve performance, but if the cost of load balancing consumes more time than is gained by better load distribution, the load balancer is not worthwhile. Typical parallel scientific applications are run for many hours, otherwise it would not be necessary to go to the trouble of implementing a parallel program to solve the problem. Therefore, load balancers can generally spend more time to achieve high parallel efficiency. However, for those cases where execution must be completed in a few minutes or less, load balancing time can become critical.

6.1.1 Centralized Load Balancing

Centralized load balancing refers to the strategy of having each executing thread synchronize at particular points in the application code. At this point, the entire object-communication graph is available to the load balancing strategy. Furthermore, the measured program performance is in a consistent state, since when all threads have reached the synchronization point, they will have performed the same number of iterations of the application algorithm. Centralized load balancing is used when the load balancing strategy requires global information about the state of the system.

Centralized load balancing does not necessarily entail a global barrier synchronization. It is centralized only in the sense that the state of the object graph is sampled at a globally consistent point. Given the design of the Charm++ load balancing framework, determining the globally consistent point requires the use of the local barrier mechanism of the framework, which implements a barrier for all objects currently residing on a particular processor. Under the Charm++ framework, a load balancer could synchronize local objects on a particular processor, distribute the local portion of the object-communication graph to load balancing objects, and then resume the local objects. When the load balancing objects had received the entire objects-communication graph, they could determine which objects to migrate, and where to migrate them. Then, possibly at the next local barrier, the migration would occur. In such design, the objects never perform a global synchronization, and load balancing objects execute while useful computation is occurring; however, it is somewhat more complicated to implement. Therefore, the centralized load balancers we implemented do use a global synchronization.

All of the centralized strategies we implemented inherit from a common Charm++ class called CentralLB. CentralLB implements the coordination and communication for various load balancing schemes, requiring a particular scheme to implement only two functions. The first of these, `QueryBalanceNow()`, lets the derived strategy say whether it needs to load balance at this synchronization point. The second, `Strategy()`, is responsible for analyzing the collected data and selecting which objects should migrate, and to where.

CentralLB receives the local barrier at-sync trigger signal from the load balancing framework on each processor and sends each processor's portion of the object graph to one processor. When this processor receives the entire communication graph, it passes the graph to the `Strategy()` function which is defined for whatever strategy has been selected. The strategy considers the data and may decide to migrate some or all objects. These decisions are returned to CentralLB, which broadcasts them to the CentralLB representative on each processor. Each processor tells the framework to migrate any local objects to their new homes. It also determines how many incoming objects it is due to receive. When all incoming objects have been received, all objects on that processor are resumed from the synchronization point, and the application continues.

RandomLB

RandomLB performs random load balancing. Although it is based on CentralLB, it does not use the object graph. It merely picks a random destination for each object. This can result in reasonable load balance if communication overhead is small and there are a large number of objects. As the number of objects per processor decreases, the load of the highest-loaded processor increases relative to the average processor load, resulting in poor parallel efficiency. Since each object is assigned to a processor by selecting a random destination, RandomLB requires a single pass through the object list, and the computational complexity of RandomLB is $O(N)$

GreedyLB

GreedyLB is an implementation of a greedy load balancing algorithm on top of CentralLB. The objects in the object-communication graph are redistributed without regard to communication or current processor assignment. A processor heap is built, so that the processor with the least assigned load is at the top of the heap. Initially, no objects are assigned to any processors, so every processor in the heap has no load, and the processor at the top of the heap is arbitrary. An object heap is also built from the objects in the object-communication graph, and arranged so that the most time-consuming object is at the top of the heap. In a loop, the heaviest unassigned object is assigned to the lightest-loaded processor, the processor's load updated and then the heaps readjusted, until each object has been assigned to a processor. For N objects, this is an $O(N \log N)$ algorithm, but since it pays no attention to current processor assignment, usually result in most of the objects being migrated.

RefineLB

RefineLB is an algorithm which improves the load balance by incrementally adjusting the existing object distribution. It refines some existing object arrangement, either the current object placement or an object placement determined by one of the other load balancers. Refinement is called with an overload parameter (typically 1.02-1.05), which governs by what factor any processor may exceed the average load. The refinement algorithm is described in Figure 6.1. Note that heaps are used to store P_{over} and P_{under} so that the heaviest and lightest loaded processors can be retrieved efficiently. The computational burden of this algorithm is low because only overloaded processors are examined. Since only a few objects are relocated to relieve overload on the heaviest loaded processors, when applied alone, this procedure also results in only a few objects being moved.

RandomRefineLB and GreedyRefineLB

RandomRefineLB and GreedyRefineLB are two strategies that apply the refinement algorithm after applying the random and greedy strategies. In most cases, we expect refinement to only produce small improvements, but for cases where the base strategy has resulted in a few badly overloaded processors, the refinement step may produce dramatic performance improvement.

RecBisectBfLB

RecBisectBfLB is a load balancer which uses the object-communication graph to recursively partition the objects until there is one partition for each processor. Initially, the communication graph is partitioned into two parts by growing regions around two arbitrary nodes (objects) in the graph. Two interleaved breadth-first searches are carried out from the two initial nodes, with whichever region has the lowest assigned load being expanded by one node at each iteration. Once two partitions are formed, each partition may be further partitioned until there is one partition for each processor. Non-power-of-two partitioning is handled by allowing weighted partitions, so to create 5 partitions, the algorithm first partitions it into pieces with loads in the ratio of 3 to 2, then the weight 3 partition is subdivided into partitions with weights in the ratio 2 to 1, etc. until finally finding 5 partitions, all with equal weights. Finally, each partition is assigned to one of the

```
threshold ← overload · averageload
P_over ← set of processors with                     Stored in a max-heap
             assigned loads > threshold
P_under ← set of processors with                    Stored in a min-heap
             assigned loads < averageload
while P_over is not empty
    p_o ← heaviest-loaded processor in P_over
    objects_o ← all objects currently               Stored in a max-heap
                 assigned to p_o
    repeat
        if objects_o is empty then
            if load(p_o) > threshold then           We are still overloaded, but
                return failure                      we have no more suitable
            endif                                   objects to move
        else
            object ← the heaviest object remaining in objects_o
        endif
        p_u ← lightest-loaded processor in P_under
        if load(p_u) + load(object) < threshold then
            deassign object from p_o and recalculate load for p_o
            assign object to p_u and recalculate load for p_u
        endif
        remove object from objects_o
    until load(p_o) < threshold
    remove p_o from P_over
endwhile
return success
```

Figure 6.1 The refinement algorithm used in RefineLB processors. This method provides good communication performance, since communicating objects are likely to be assigned to the same processor, but under certain conditions, may do a poor job of balancing computation, since each partition does not have the freedom to fine-tune its load by selecting some unconnected objects. This algorithm is similar the an algorithm used by Farhat [45] to partition finite element meshes.

MetisLB

MetisLB uses the METIS graph partitioning library [46], [53] to partition the object-communication graph. METIS uses a multilevel k-way hypergraph partitioning algorithm to partition the graph, taking into account node (object) weights and communication loads. First a graph is successively "coarsened" by treating groups of nodes as single nodes. Then, starting from the coarsest graph, the graphs are partitioned and refined by one of several algorithms, until at the finest level, a partitioned version of the original graph results. METIS is designed to partition graphs of hundreds of thousands to millions of nodes with typical graph degrees of approximately 10, which commonly occur in applications such as finite element mesh generation. Since our graphs typically contain only a few hundred or thousand nodes, often with higher degrees, the partitionings produced by METIS may not always work as well as simpler methods. MetisLB takes the graph produced by CentralLB and translates it to a form understandable by the METIS library for partitioning. The processor assignments recommended by METIS are then translated back into the format used by CentralLB for migration.

6.1.2 Neighborhood load balancing: NeighborLB

Another approach to load balancing is to reduce the message traffic and synchronization overhead by performing neighborhood load balancing. This is implemented in the NeighborLB strategy. Rather than collecting the entire object-communication graph on a single processor, each processor sends summary statistics about its load to a *neighborhood* of processors. This summary data includes information such as object loads, overhead load, idle time, and total wall and CPU time consumed by the process during the last load balancing period, and relative processor speed. When each processor has received data from all its neighbors, it compares its load to the average of its neighbors loads. If it is below the neighborhood average, it tells its neighborhood that it does not need to migrate any objects away. If it is overloaded, it repeatedly selects the largest of its objects which can be passed to the lightest loaded neighbor without overloading the neighbor, until the processor load drops below the neighborhood average, or none of the processor's objects will fit on any of the neighbors without overloading them. Each neighbor is informed which objects will be arriving, and then the load coordinator is asked to move those objects away. When all elements have arrived, the NeighborLB strategy asks the load coordinator to resume the synchronized objects, and execution continues. NeighborLB currently uses a neighborhood of four objects, but this is easily modifiable to allow bigger neighborhoods.

6.2 Load Balancer Performance

To evaluate the performance of our load balancers, we implemented a test program to simulate typical parallel applications, but with greater control of the computation and communication. The `lb_test` program is described below, followed by performance evaluations of the Charm++ load balancers we implemented.

6.2.1 The `lb_test` Program

We designed `lb_test` to simulate the performance of typical parallel programs. The program builds an array of test objects, which communicate with each other using one of several communication patterns. Each test object sends messages to its neighbors, awaits messages from those neighbors, then performs computation for some period of time. This process is repeated a number of times, with a periodic local synchronization to allow load balancing, and execution time per iteration is periodically reported.

The dummy computation load used by `lb_test` is a small loop that performs several floating-point operations, and accumulates the result in such a way that compilers do not optimize the loop away. Each object is assigned an amount of computation to perform during each iteration. Each object is assigned a random amount of work at program start, using a random distribution selected at compile time, and minimum and maximum limits given at run-time. Several distributions were tried, with an x^4 distribution providing a sufficient spread of object times to produce unbalanced performance for the initial object distribution. At program start, the computation time is calibrated by timing one hundred iterations of the loop. That timing is used to calculate the initial estimate of iterations per second, I_0. Since only a small number of iterations were timed, it is likely that poor timer resolution and timer overhead will produce inaccuracy in I_0. Therefore, its value is corrected by timing the execution of I_0 iterations of the loop, and calculating a more accurate I_1. This procedure is repeated, and the result is used to compute the number of iterations the object will perform during each computation loop.

At run-time, one of several communication topologies may be selected, to study how different load balancers handle different communication patterns. The topology with the fewest messages is `ring`. Each object i sends one message to object $(i+1)$ mod *num_objs*, and receives one message from $(i+1)$ mod *num_objs*. The second algorithm `mesh2d`, builds a square 2-D mesh of objects, leaving a portion of the mesh unfilled if the requested number of objects is not square. Except for border elements, each object sends and receives messages with its four neighbors in the mesh. The third algorithm, `randgraph`, builds a random graph connecting the objects. At compile time, we select a ratio of communicating links to the total links in a completely connected graph. Message sizes are fixed, but can be changed at compile-time.

Load balancer performance is measured by observing how long is required for an arbitrarily selected object requires to complete a number of iterations before and after load balancing. Since the work function is calibrated with wall-clock time, we can calculate speedups by comparing the total computation time of all the objects (ignoring communication) to the actual times, including load imbalance, idle-time resulting from unavailability of the incoming messages, and communication overhead.

6.2.2 Performance Results

Several of the load balancer strategies were tested using `lb_test` on the ASCI-Red computer at the Sandia National Laboratories. The ASCI-Red is a collection of 9,536 Intel Pentium II Xeon processors connected by a high-speed mesh interconnection network. Seven load balancers were tested: RandomLB, RefineLB, GreedyLB, RandomRefineLB, GreedyRefineLB, RecBisectBfLB and NeighborLB. MetisLB could not be tested due to limited single-processor memory of the ASCI-Red.

All speedups are compared to the raw computation time, so single-processor speedups can be less than one, due to parallelization overhead.

The first experiment examines load balancer performance on 2 to 64 processors. The experiment used 512 object with the `mesh2d` communication graph, with computation times ranging from $100\mu s$ to $4100\mu s$, for a total time of 0.45s, a mean computation per object of $870\mu s$, and a grainsize of $0.43\mu s$ per message. Note that, for all of the experiments, since object grainsize includes both local and off-processor messages, the processor granularity will vary depending on the number of processors and object placement. Figure 6.2 shows the performance of the load balancer for the four basic strategies, GreedyLB, NeighborLB, RandomLB, and RecBisectBfLB. No load balancer produced linear speedups, due to the small grainsize. GreedyLB and NeighborLB produced similar efficiency for all numbers of processors. One might expect RecBisectBfLB to produce better performance, since it minimizes off-processor communication. However, because RecBisectBfLB only considers partitions of objects that are connected in the object-communication graph, it does not always balance the computation as well as the other methods. In this case, it must be object overhead (relative to the small computation time), not message overhead that results in poor speedup. Therefore, since RecBisectBfLB does not balance the computational load as well its performance suffers.

Figure 6.3 compares GreedyRefineLB with GreedyLB and RefineLB. All produced similar efficiencies, with GreedyRefineLB performing somewhat better for 64 processors. This indicates that GreedyRefineLB was able to fix some slight imbalance left by the greedy algorithm alone, decreasing the load of a few slightly-overloaded processors.

Figure 6.4 compares RandomRefineLB with RandomLB and RefineLB. As expected, RandomLB produces poor performance. Since speedup is determined by the highest loaded processor, for a random distribution of random-sized objects, the standard deviation of the processor load increases as the number of objects per processor decreases. Consequently, as the number of processors increases for a fixed number of objects, the number of objects per processor decreases, and the load on the heaviest-load processor increases relative to the average load, decreasing parallel efficiency. Both RandomRefineLB and RefineLB produced similar efficiencies, indicating that for this case the randomization of objects does not contribute significantly to improving the performance of the

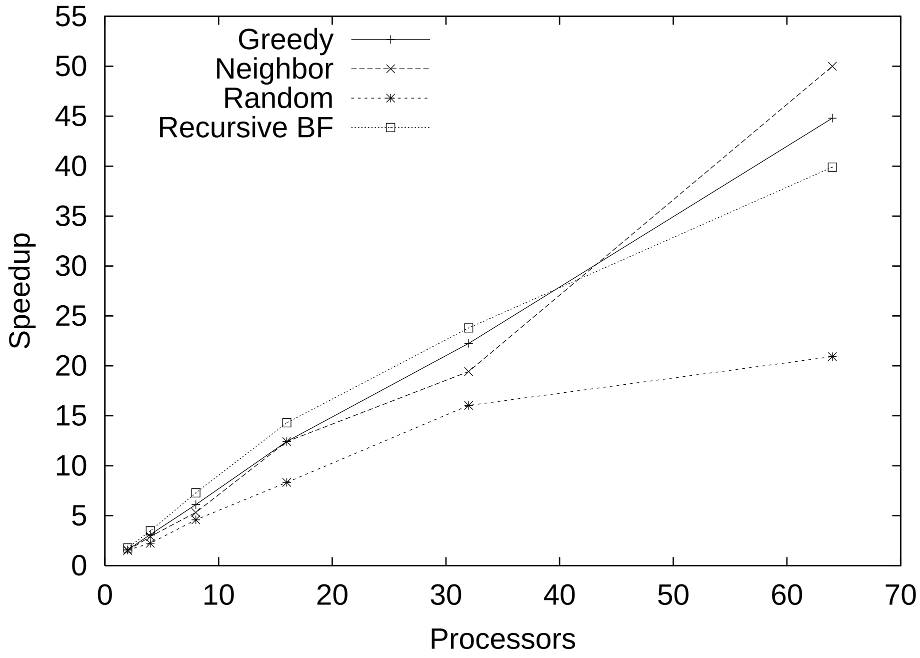

Figure 6.2 Speedup for load balancers on the `lb_test` benchmark for experiment 1

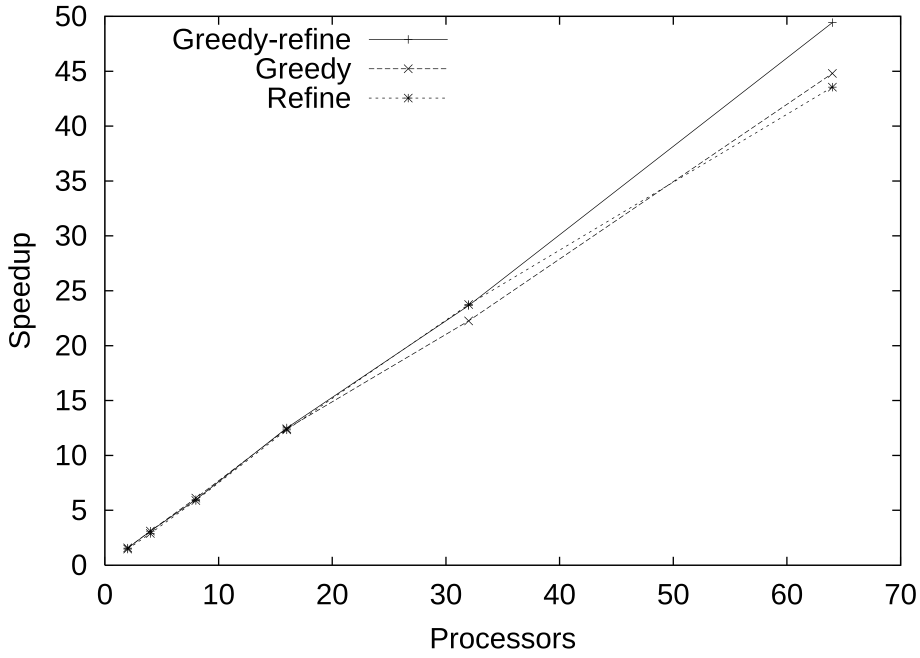

Figure 6.3 Speedup for Greedy load balancers on the `lb_test` benchmark for experiment 1

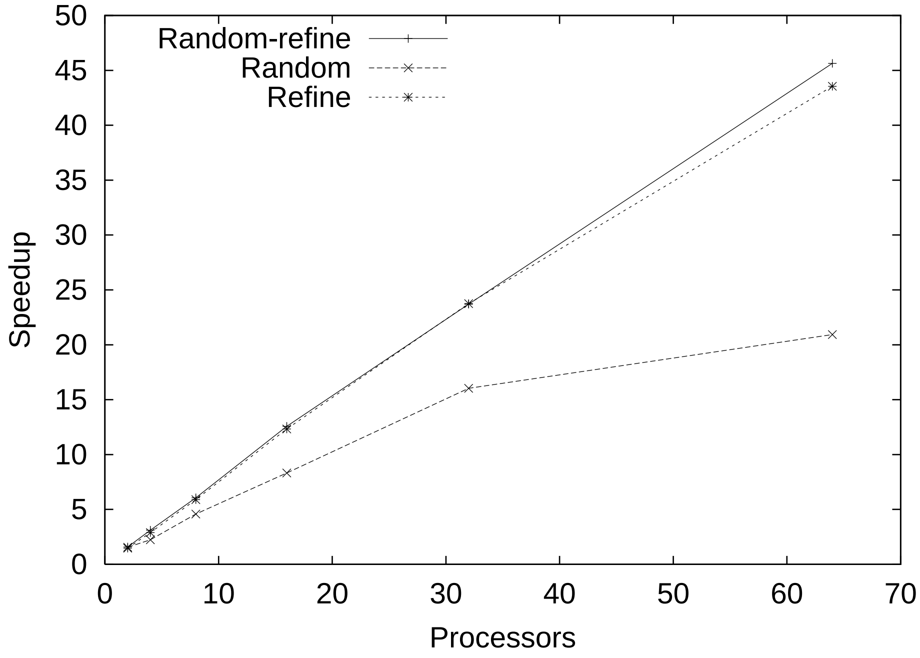

Figure 6.4 Speedup for Random load balancers on the `lb_test` benchmark for experiment 1

refinement algorithm.

The next experiment examines the load balancer performance for larger parallel machines. The experiment was performed using 64 to 1024 processors, and the number of objects was increased from 512 to 4096, again with the `mesh2d` communication graph. Total computation time was also increased, with object times ranging from 500μs to 4500μs, for a total time of 5.3s, a mean computation per object of 1.3ms and a grainsize of 330μs/message. Figure 6.5 compares GreedyLB, NeighborLB and RandomLB. Again, as expected, RandomLB produces poor performance. Both GreedyLB and NeighborLB show similar performance for up to 512 processors, corresponding to an average of 8 objects per processor. For 1024 processors (4 objects/processor), NeighborLB performs much worse. The most likely explanation for this is that a small number of processors got stuck with 2 or 3 computationally-expensive objects. Those objects could not be given to a neighbor because all the neighbors were close to the average load, and the strategy will not intentionally overload a neighbor processor. Therefore, that processor remained overloaded. This suggests a future improvement for NeighborLB, to allow a processor to occasionally overload a neighbor if the processor has failed to relieve its own overload after several load balancing cycles.

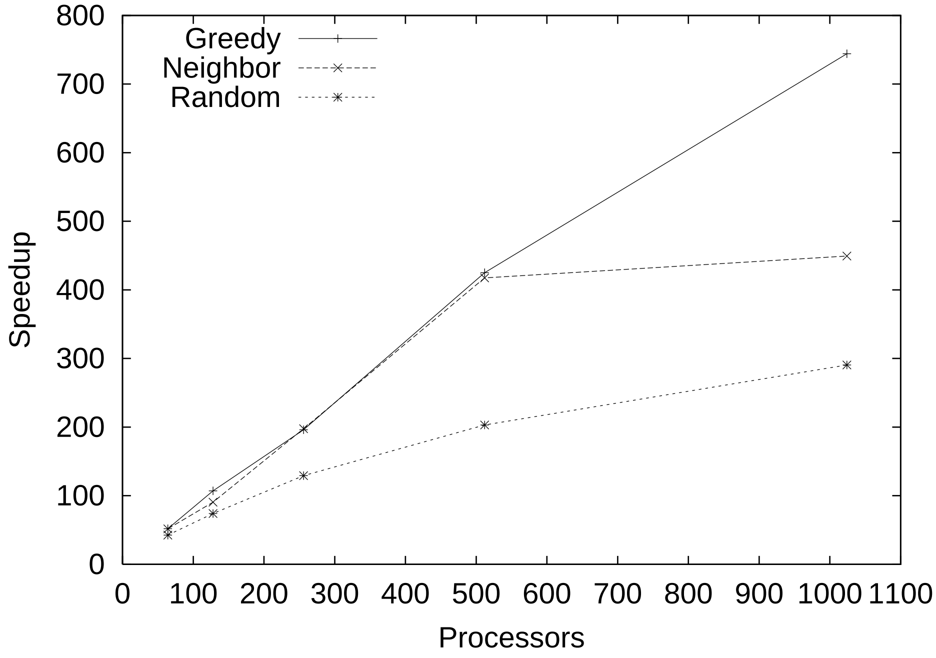

Figure 6.5 Speedup for load balancers on the `lb_test` benchmark for experiment 2

Figure 6.6 shows the performance for the Greedy load balancers for the second experiment. All perform equally up to 512 objects per processor. For 1024 processors, GreedyLB produces good efficiency, whereas GreedyRefineLB and RefineLB both perform poorly. Since GreedyRefineLB should produce at least as good a load balance as GreedyLB alone, the poor performance of GreedyRefineLB must result from load behavior ignored by the refinement algorithm. One probable explanation is that, in attempting to relieve a small amount of overload on a number of processors, the refinement algorithm places a number of low-computation objects on some processors. Although these objects do not add much computation to the processor, each does add communication, which is ignored by the refinement procedure. Modifying RefineLB to perform some simple communication modeling would remedy this defect.

Figure 6.7 examines the performance of the RandomLB strategy for the second experiment. As in the previous cases, RandomLB alone does poorly. RandomRefineLB and RefineLB are again similar for 512 processors. For 1024 processors, RandomRefineLB has much lower efficiency than RefineLB alone. This suggests that by randomly shuffling the objects has exacerbated the small-object problem, causing the refinement algorithm to decrease the load on many objects by moving

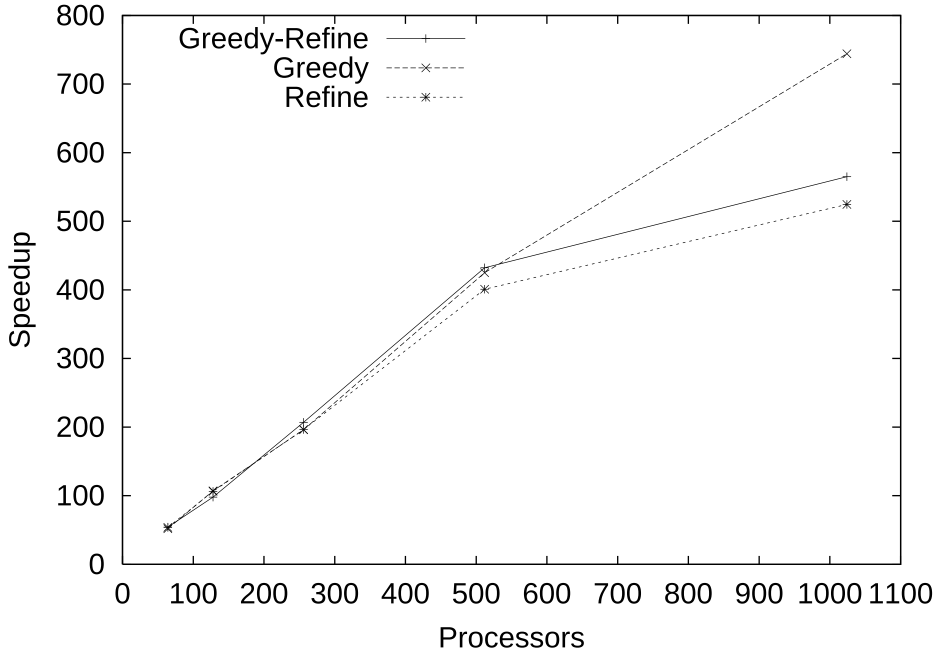

Figure 6.6 Speedup for Greedy load balancers on the `lb_test` benchmark for experiment 2

low-computation objects rather than computationally intensive ones.

The third experiment was for a large machine with a large amount of computation. The test used 4096 objects, with computation times ranging from 1ms to 41ms. Total computation per iteration was 37.1s, with a mean computation per object of 9ms. The `mesh2d` communication topology was used, for a grainsize of 2.3ms/message. Figure 6.8 compares the performance of GreedyLB, NeighborLB, RandomLB, and the initial distribution (None), which is a round-robin placement of objects. In this case GreedyLB produced near-linear speedups, indicating that communication costs were negligible for these selected communication parameters. NeighborLB produces similar speedups for 256 processors, but is less efficient on more processors. This is explained, as in previous cases, by a few processors being unable to migrate away extra load because they have a few computationally-intensive objects that are too large to give to any neighbors. The round-robin initial distribution performs better than RandomLB, showing that RandomLB must be overloading a few processors by giving them too many objects.

Both GreedyLB and GreedyRefineLB performed well in this experiment, as shown in Figure 6.9. RefineLB performed well for up to 512 processors, but was less efficient for 1024 processors. Since

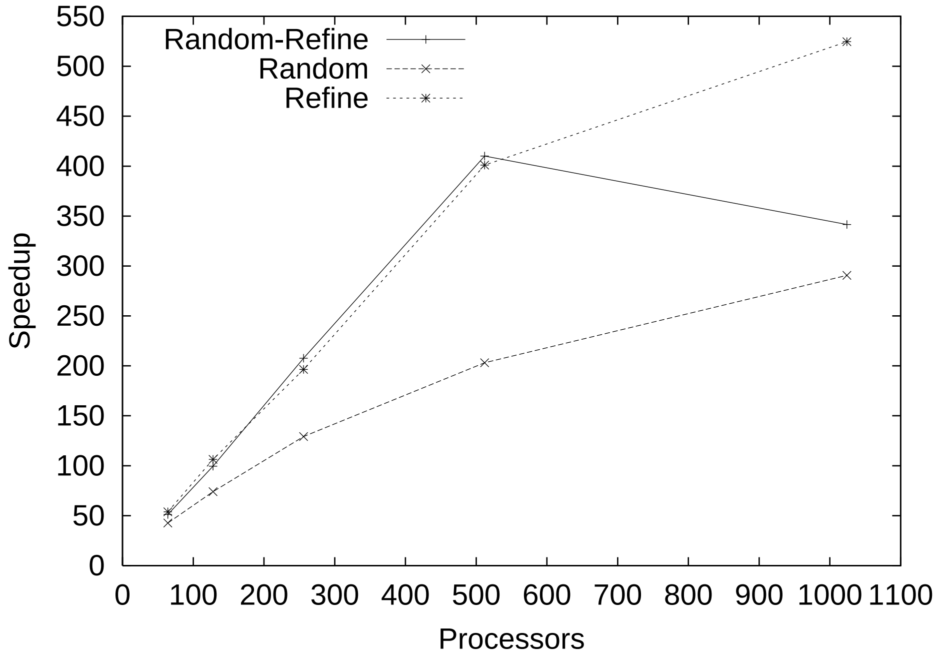

Figure 6.7 Speedup for Random load balancers on the `lb_test` benchmark for experiment 2

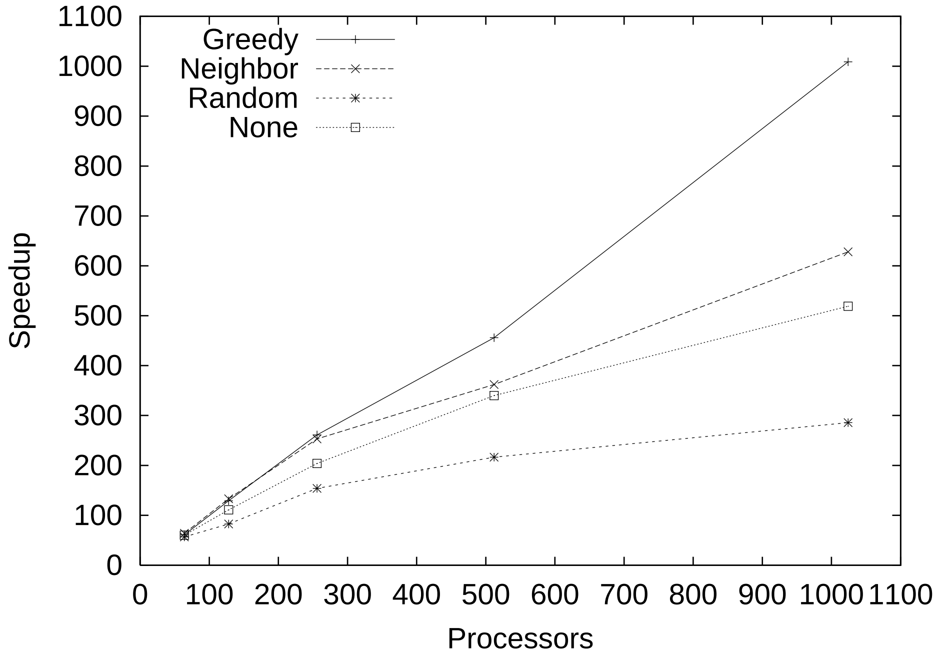

Figure 6.8 Speedup for load balancers on the `lb_test` benchmark for experiment 3

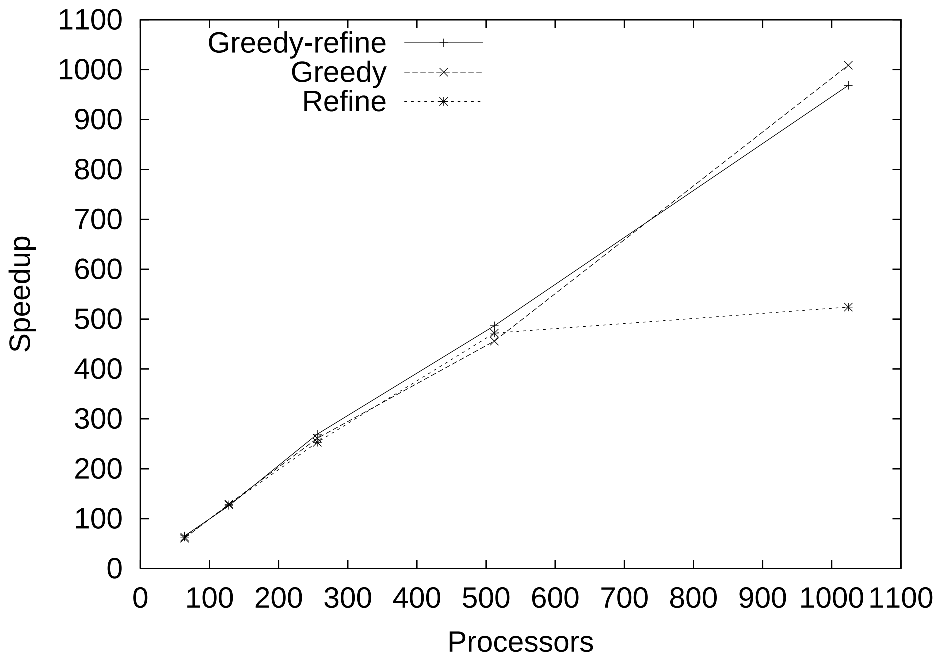

Figure 6.9 Speedup for Greedy load balancers on the `lb_test` benchmark for experiment 3

communication costs are negligible in this experiment, and since GreedyRefineLB did well, the argument about refinement overloading a few processors with many small objects is unlikely. A more likely explanation is that RefineLB was only able to improve slightly on the existing initial distribution.

In Figure 6.10, RefineLB and RandomRefineLB both performed well for up to 512 processors, but achieved a speedup of only about 500 for 1024 processors. As hypothesized above, neither load balancer is able to completely balance the load when starting with a poorly-balanced distribution, but in this case, refinement did improve the load balance over RandomLB alone.

The next experiment examined the load balancer performance in the presence of heavy message traffic. Computational parameters were selected to be the same as in the first experiment, but the message size was increased from 1024 bytes to 100,000 bytes. Figure 6.11 shows the results of this experiment. As expected, speedups are low due to the heavy communication penalty. Nevertheless, RecBisectBfLB performs better than either GreedyLB or GreedyRefineLB, showing that RecBisectBfLB is better able to compensate for heavy communication.

Most of the load balancing strategies we implemented ignore the current distribution of objects

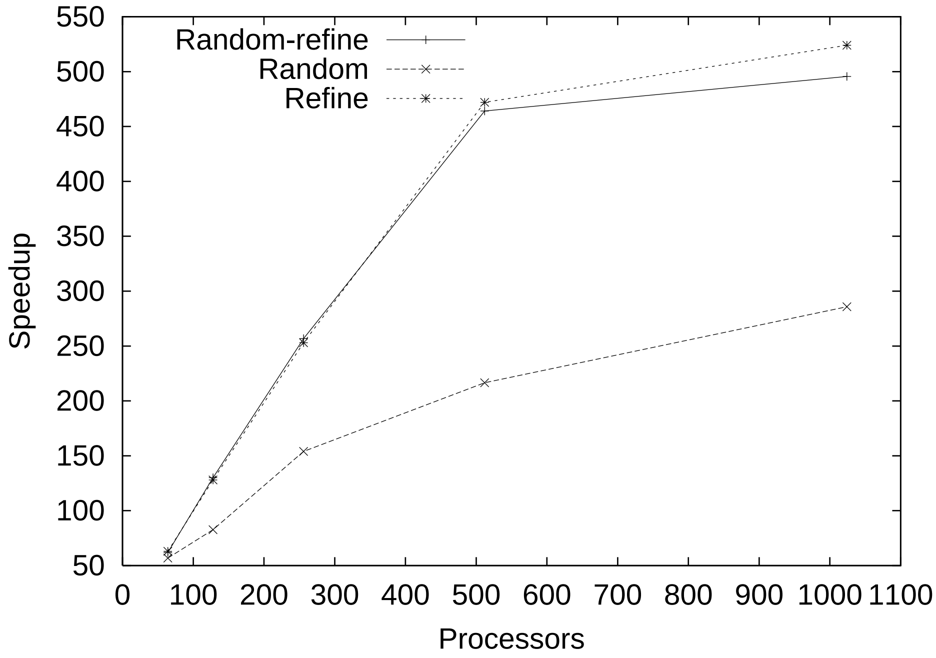

Figure 6.10 Speedup for Random load balancers on the `lb_test` benchmark for experiment 3

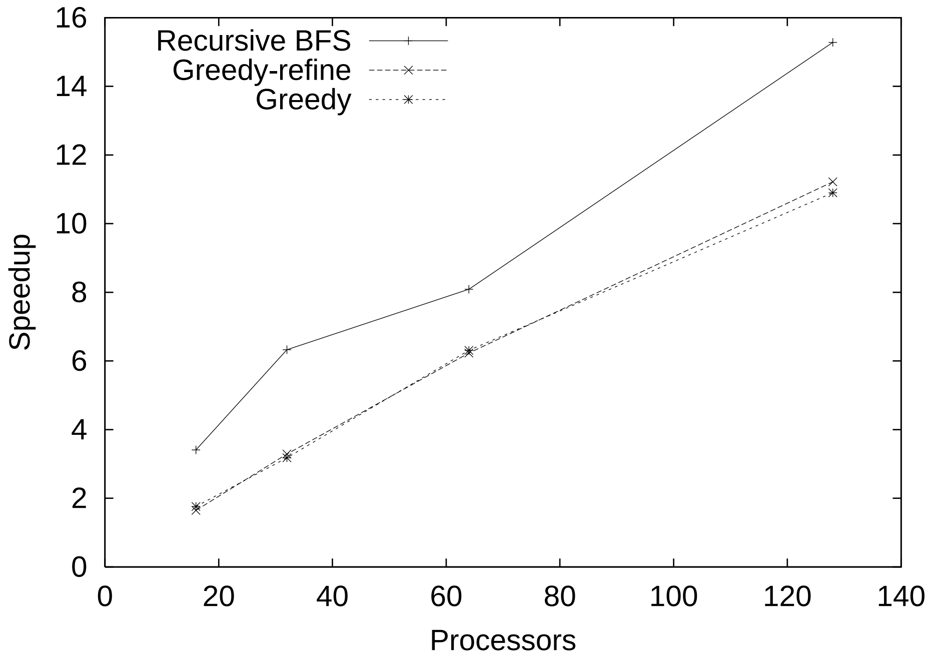

Figure 6.11 Speedup for load balancers on the `lb_test` benchmark for experiment 4 (100000 byte messages)

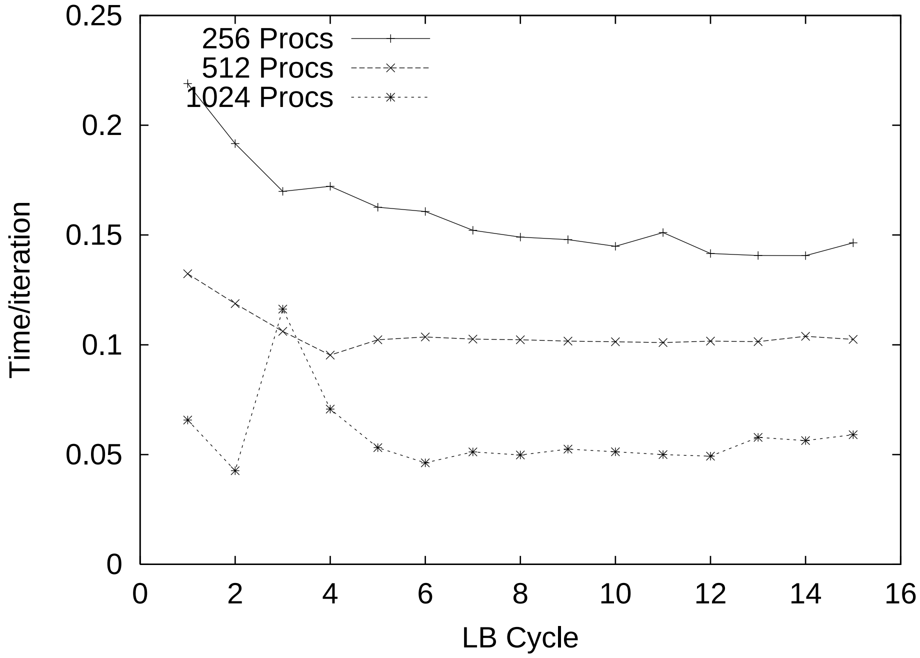

Figure 6.12 NeighborLB performance for successive load balancing cycles

on processors. Only RefineLB and NeighborLB use the current object placement as the starting point for the load balancing procedure. RefineLB did not show any progressive improvement in speedup for successive load balancing cycles, but NeighborLB seemed to require a few load balancing cycles to achieve its best load balance, particularly for larger numbers of processors. This effect is displayed in Figure 6.12, which shows the behavior for the third experiment. For the various numbers of processors, neighborhood load balancing takes about 6 steps to converge to the steady-state time per iteration. One way to decrease this time would be to expand the size of the neighborhood, allowing objects to move to unloaded processors faster. There is a tradeoff, however, in that a larger neighborhood will require a longer time for load balancing, possibly offsetting the time saved through more efficient load balancing.

Figure 6.13 shows the time required for load balancing on processor zero (where CentralLB does its collection), for various load balancing strategies. The graph shows the time for executing the load balancing strategy computation, for 4096 objects running on 64 processors with a 2D mesh connection network. As expected, RecBisectBfLb is the most computationally-intesive strategy, followed by GreedyLB and GreedyRefineLB. The others all exhibit very small computation times.

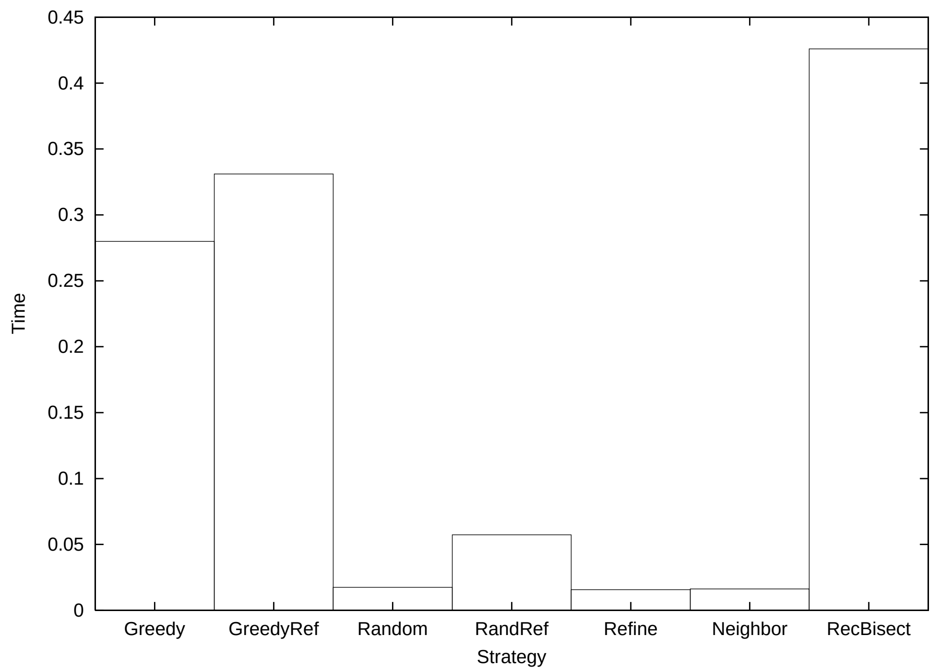

Figure 6.13 Time consumed for load balancing by various algorithms

However, even for the slower strategies, computation times this small are not the dominant factor in determining load balancing speed. Unfortunately, the dominant cost associated with load balancing is not the computation time, but the idle time spent waiting for the load data from other processors, which fluctuates randomly depending on when messages are delivered to the load balancer. This cost increases even more when the processors are badly balanced, because the underloaded processors will waste more time waiting for the overloaded ones to send their data. This suggests that strategies (especially centralized ones based on CentralLB) would benefit from a multi-phase load balancing scheme which overlaps load balancer communication with ongoing computation. In such an implementation, the data collection messages are sent at one local barrier, and the objects immediately resumed. Before the next load balancing barrier, the strategy makes its migration decisions and sends them to all the processors. When the next local barrier arrives, the migration decisions are put into effect, and execution resumes as soon as migration is complete.

To examine load balancing behavior in another application, we performed another experiment using the `jacobi` benchmark described in Chapter 3. The simulation is a 2048×2048 grid, distributed evenly over a 32×32 array of objects.

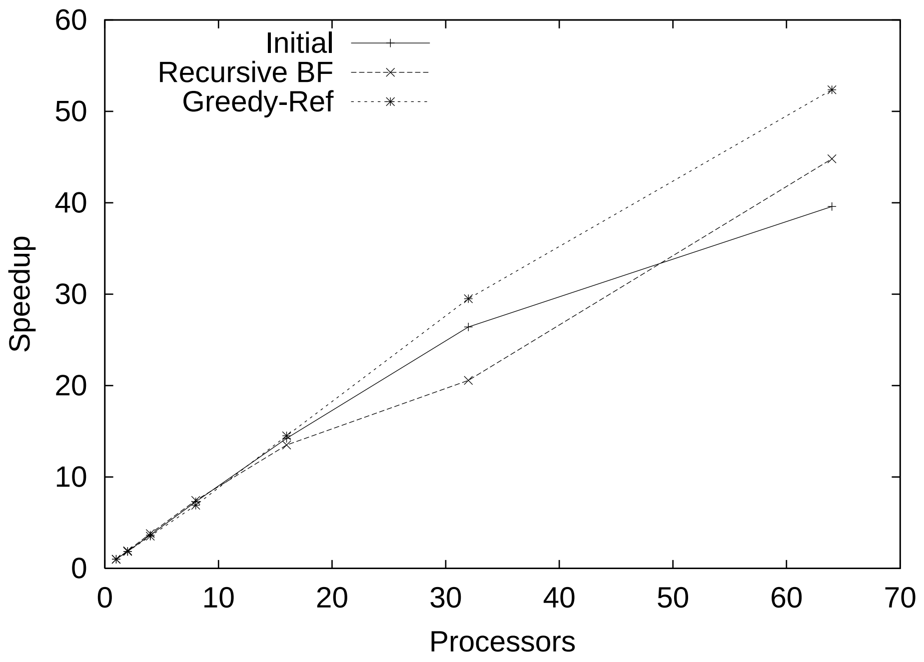

Figure 6.14 Load balancing for the `jacobi` benchmark.

Figure 6.14 shows the performance of the program after load balancing for the GreedyLB and RecBisectBfLB strategies, as well as the initial object distribution. Since all of the objects are equal-sized, and there are many objects per processor, GreedyLB does well. The initial, round-robin distribution also does well for up to 32 processors, since for those cases, the systematic distribution results in allocating the grid in columns. For 64 processors, however, the initial distribution results in no object being on the same processor with any of his neighbors, which results in more communication than GreedyRefineLB, where random clustering results in some accidental communication optimization. The RecBisectBfLB does not do quite as well as the other strategies on 32 processors, but this is probably due to one or two partitions containing a slight amount of extra work.

6.2.3 Other applications

In addition to `lb_test` and `jacobi`, the load balancing framework has been used with several other applications, including CohesiveFEM (described in Chapter 3), a C++-based finite element code implemented using a Charm++ finite-element simulation framework, and a version of ROCFLO, a fluid-dynamics code. The ROCFLO code uses a clever combination of Charm++ object arrays, Converse threads, and an MPI-style communication library to convert the original Fortran/MPI application into a migratable-object program, where virtual processors are simulated by Converse threads, and each thread is controlled by an object array element.

6.3 Other Load Balancing Challenges

We have also implemented several of the features needed to utilize non-dedicated workstation clusters in the NeighborLB strategy.

6.3.1 Heterogeneous Clusters

The NeighborLB load balancer takes processor speed into account when load balancing. Each processor includes the processor speed in the data it sends to its neighbors. Whenever a processor considers moving an object to an under-loaded neighbor, the processor scales the object's time to determine how much load (execution time) it will add to the neighbor. Only if migrating a particular object will not result in overload, given that the neighbor may require more time to execute that object, will the migration proceed.

6.3.2 Vacating a Processor

We have also implemented the vacating a processor feature as part of the NeighborLB strategy. Should a a particular processor enter the vacate state, at the next load balance trigger, it will inform each neighbor that it needs to vacate itself. The neighbors use that information to ignore the vacated processor in calculating the neighborhood load average. Furthermore, objects are never migrated to the vacated processor. Meanwhile, the strategy on the vacated processor migrates all of its migratable work to its neighbors.

When a vacated processor receives a signal to resume computation, it informs its neighbors at the next load balance trigger. They immediately begin including the reactivated processor in the neighborhood average computation, and since the reactivated processor will most likely be idle, it will be selected first for re-population. If several neighbors each contribute load to the reactivated

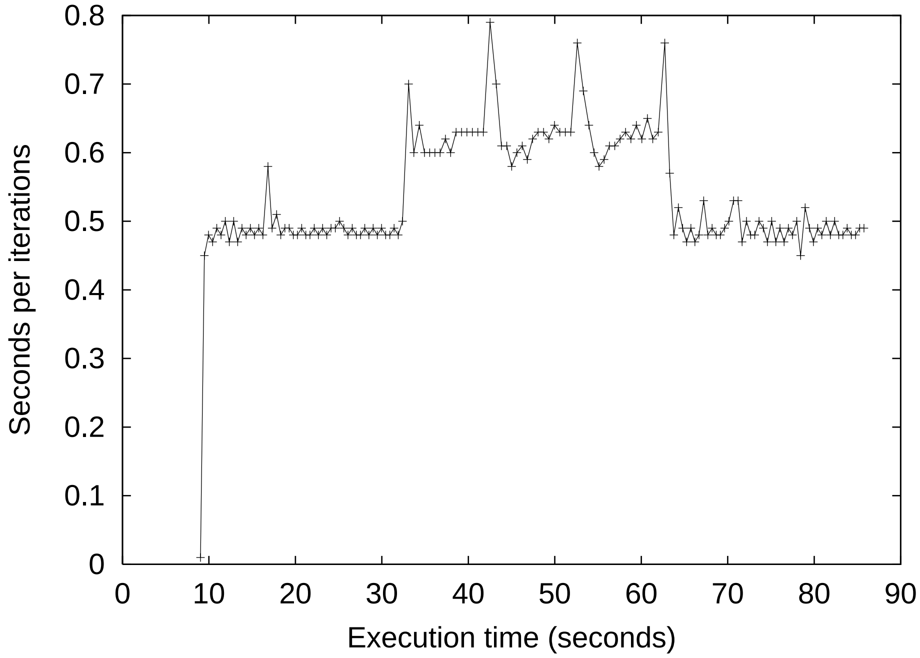

Figure 6.15 Execution time (seconds per iteration) for the `lb_test` benchmark, when the system is asked to vacate a processor, and then allowed to return work to that processor.

processor, that processor is likely to be temporarily overloaded, but such overloading is remedied in future load balancing cycles.

Figure 6.15 shows the response of the load balancer to a request to vacate a processor while running the `lb_test` benchmark. This test was run on four Sun Ultra 10 desktop workstations which were otherwise idle. Approximately 30 seconds after program start, the load balancers is told to vacate one of the processors. As expected, execution time per step increases in response to the loss of one CPU. For the particular problem, communication is not a signifigant factor, so the decreased communication resulting from vacating a processor can be ignored. Initially, each iteration requires about 0.48 s, for an ideal sequential time per step of 1.9 s. After vacating a processor, only three processors are available, so perfect scaling should produce a time per step of 0.64 s, which matches the experimental results. Viewing the process utilization on the vacated node shows that it is using a negligible portion of the available processor time. At 60 seconds, the load balancer is directed to resume using the vacated processor, and the execution time returns to its initial level.

6.3.3 Background Load

The Jacobi example demonstrated in Chapter 3 demonstrates the disproportionate penalty background load can impose on some applications. The object-array migration scheme allows applications to maximize the throughput on systems with background load.

Consider the steps required to maximize throughput for the Jacobi example. First, using the facilities available in the Converse run-time library, we obtained a measure of processor availability on each processor: this is essentially the CPU time obtained by a process during the time interval, divided by the elapsed time. Converse can also provide information about when the parallel program on that workstation was idle waiting for messages. Subtracting CPU utilization and idle time from elapsed time gives an estimate of the CPU time consumed by outside processes. Based on this information, the system decides when there is a severe load imbalance, and calculates what fraction of the load must migrate to other processors. During the run, the system automatically keeps track of the time consumed by each array element. At migration time, the system selects a set of elements for migration which most closely match the desired fraction of the total load. Currently, migrated elements are distributed evenly to the other processors.

Figure 6.16 shows the performance for the `lb_test` benchmark on four Sun Ultra 10 workstations. The load balancing interval was set artificially high (approximately 15 seconds between load balancing opportunities) so that the effect of background load before migration would be recorded. At about 25 seconds into the run, an external job was started on one of the processors. Execution time per iteration immediately doubles, due to the data dependencies in the benchmark. The load balancer senses the performance drop approximately 35 seconds after program start, and migrates some of the work off of the overloaded processor. After about 50 seconds, the background job is killed. No immediate change in execution time is apparent, because even though the processor is fully available again, the load balancer has not yet had the opportunity to sense this condition. After 60 seconds, the load balancer detects the change in background load, moves work back to the processor, and the initial execution time per iteration is restored.

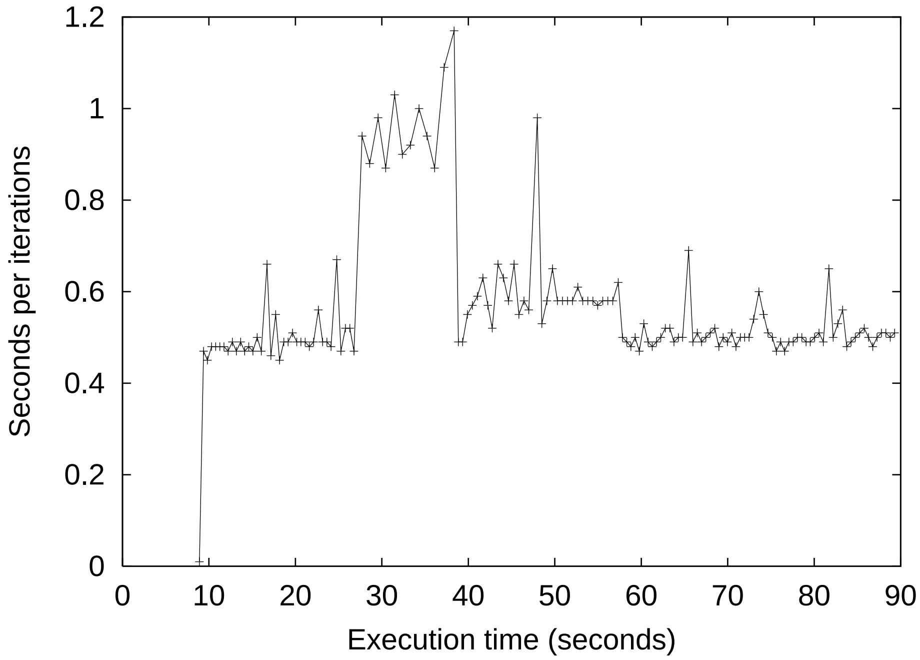

Figure 6.16 Execution time for `lb_test` benchmark, where the system detects background load, and migrates work away, and later determines that the background load has ceased, so returns work to the processor.

CHAPTER 7

CONCLUSION

In this dissertation, we have examined the problems of automatic load balancing, and developed an approach for performing automatic load balancing for a variety of applications.

We examined problems encountered in load balancing. Application imbalance may appear as initial imbalance, slow variation in application load, which results in a gradual degradation in program performance unless corrected, and fast variation in application load, which results in immediate performance degradation which persists until corrected by rebalancing the load. External imbalance arises on non-dedicated parallel machines such as workstation clusters. Background tasks may interfere with a parallel program, resulting in disproportionate decrease in parallel efficiency until the load imbalance is corrected. Workstation networks present other challenges such as variation in processor speeds, and limited availability, which may require a processor to stop consuming more than a fraction of the available CPU time until the workstation's owner allows the processor to resume.

The Charm++ system, with its data-driven execution, is particularly well suited to load balancing with migratable objects and load measurement. The new Charm++ object type, object arrays, simplifies the creation of object-based applications. Object arrays include mechanisms for supporting migration of elements from processor to processor, including a message forwarding mechanism, and reductions which function efficiently with migrating objects.

Our approach has been embodied in a load balancing framework, which provides a general purpose interface for applications. The framework provides the mechanisms for object measurement, communication tracing, and synchronization primitives to simplify object migration. It also provides a load balancing strategy interface, so that different load balancing strategies can access the load information. By separating applications from strategies, a single application may be tested with several different load balancers. Similarly, a particular load balancing strategy can be evaluated with a variety of applications.

Next we described NAMD, a production-quality molecular dynamics program which uses object based load measurement and object migration for effective load balancing. NAMD is a Charm++ program, which uses the framework for load balancing, but handles object migration within the application. Our experience with NAMD demonstrates that load measurement is an accurate and efficient way of measuring load, and migrating work in the form of objects is effective in regulating load. It also showed that implementing all the mechanisms for load balancing is a complex, time consuming step in implementing a parallel application.

A variety of load balancing strategies were developed that leverage the capabilities of the load balancing framework. The strategies aim at solving a multidimensional optimization problem involving computation and communication costs, and are necessarily heuristic in nature. These strategies were tested under various conditions with several applications, to understand the conditions under which individual strategies might be suitable. Finally, we described how the framework and strategies support the capabilities required for workstation-cluster environments.

7.1 Future Work

The Charm++ object arrays extension and automatic load balancing using our object framework have already proved valuable to a number of projects. However, several improvements and areas for further research deserve future study:

New object collections: Allowing users to create a wider variety of collections of objects (e.g. trees) would simplify the implementation of certain algorithms. Making arrays dynamically re-sizable would also be useful in certain applications. The framework is flexible enough to handle such new constructs.

Advanced communication: The multicast system needs to be completed. It currently supports only broadcasts to all members of the array. Some applications would benefit from a complete implementation of multicast to any subset of an array. Implementing some form of proxy array, generalizing the proxies used in NAMD, would be another important addition. A general purpose proxy would allow array elements to periodically publicize some information about themselves on all processors where that information was needed. The proxy objects could then collect information from the proxy clients at the destination processors, and return that information to the source object. The run-time system would determine where proxies were needed, based on where the client objects were currently located.

Fortran support: Several of the applications we have ported to object arrays have been Fortran/MPI applications. We already have a preliminary MPI emulation library for the common calls used in such applications. The most time-consuming remaining part of porting such applications is writing the pack and unpack routines for lengthy lists of Fortran global variables. For migration efficiency, and for economical memory use, those variables should be separated into three categories: read-only variable that are initialized at program start, which can be stored on each processor so that they never need to be migrated, temporary variables such as loop counters, which never hold useful data at migration times, and actual state variables which must be packed to migrate the object. A pack-function preprocessor could automatically analyze such Fortran programs to determine which variables fall into which categories, and create customized efficient pack and unpack functions.

Migratable threads: Another approach to porting MPI programs is with user-level threads. When an MPI call blocks, only the calling thread blocks. There are multiple such threads on each processor, which can be migrated using our framework. This approach is less intrusive, in that it requires fewer changes to the MPI program. A primitive version of migratable Converse threads has already been used in the ROCFLO application. In ROCFLO, each thread is owned by an object array element, so that when the array element moves, it packs its thread's state, migrates, and recreates the thread state on the new processor. A cleaned up, system-level version of migratable Converse threads, using the load balancing framework for dynamic load balancing, is feasible, and would be a useful addition to the Converse run-time system.

Communication logging: In addition to adding communication operations to object arrays, the load balancing framework needs to be extended to include logging for these types of operations, so that load balancing strategies can recognize those communication patterns. For example, consider how a load balancer should handle a multicast to a number of objects. Optimal communication performance would be obtained by placing those objects on the sender's processor, but load considerations may prohibit that. If the load balancer sees the multicast as several object-to-object messages, it would not realize that by placing all of the receiving objects together on the same processor, one message would suffice to provide the data to all of the objects.

NAMD: Once more complex communication logging is added to the framework, NAMD's load balancer will be modified to use framework data exclusively for load balancing decisions. After making NAMD's load balancer completely application-independent, it will be moved into the pool of generally-available load balancers, available for use with other applications.

Execution trace logging: The framework currently stores only object times and message volumes, minimizing the memory used by the load database. However, some interesting load balancing strategies [77] may need to build the complete object dependency graph, requiring complete trace information (object A sent N bytes to object B at time T, etc.). Using the existing interfaces, the load coordinator can collect such information, but storing such information, and disposing of old trace information when memory becomes scarce.

New strategies: Since the framework encapsulates most of the mechanisms of load balancing, the effort required to implement new strategies is substantially reduced. Strategies which model communication better would seem to offer the best opportunity for achieving better parallel efficiency for more difficult problems. We have already experimented with a branch and bound load balancer using simulation [78], which should prove useful in real parallel applications.

Cooperative machine sharing: An interesting application of the standard application model specified by the framework is to allow a number of Charm++ applications running on a parallel computer to coordinate the sharing of processors among the applications. Consider two programs running on a single parallel machine. Application 1 runs most efficiently on P processors but can take partial advantage of additional processors. Application 2 will utilize as many processors as are available. Initially, both applications are running. Each application's load balancing strategy communicate with a central machine scheduler, which directs application 1 to use P processors, and gives the rest to application 2. When application 2 finishes, it reports to the scheduler, which can then direct application 1 to use the rest of the processors, since they would others be idle. If application 3 should start running, the scheduler will direct application 1 to vacate some processors to allow application 3 to run.

We believe that we have demonstrated the promise of the automatic object-based load balancing approach. We expect that this approach will form the basis of future implementations of highly complex and adaptive parallel applications.

REFERENCES

[1] David E. Culler, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Brent Chun, Steven Lumetta, Alan Mainwaring, Richard Martin, Chad Yoshikawa, and Frederick Wong, "Parallel Computing on the Berkeley NOW," in *9th Joint Symposium on Parallel Processing, Kobe, Japan*, 1997.

[2] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer, "BEOWULF : A parallel workstation for scientific computation," in *International Conference on Parallel Processing, Vol.1: Architecture*, Boca Raton, USA, Aug. 1995, pp. 11–14, CRC Press.

[3] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, May 1994.

[4] L. V. Kalé, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon, "Converse: An Interoperable Framework for Parallel Programming," in *Proceedings of the 10th International Parallel Processing Symposium*, Honolulu, Hawaii, April 1996, pp. 212–217.

[5] L.V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on C++," in *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, September 1993.

[6] Sanjeev Krishnan and L. V. Kalé, "A parallel adaptive fast multipole algorithm for n-body problems," in *Proceedings of the International Conference on Parallel Processing*, August 1995.

[7] D.L. Eager, E.D. Lazowska, and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering*, vol. 12, pp. 662–675, May 1986.

[8] Songnian Zhou, "An Experimental Assessment of Resource Queue Length as Load Indicies," Tech. Rep. UCB/CSD 86/298, University of California, Berkeley, June 1986.

[9] Songnian Zhou, "A Trace-Driven Simulation Study of Dynamic Load Balancing," Tech. Rep. UCB/CSD 87/305, University of California, Berkeley, September 1987.

[10] Will E. Leland and Teunis J. Ott, "Load-balancing Heuristics and Process Behavior," in *Performance '86 and ACM SIGMETRICS Conference*, Raleigh, North Carolina, May 1986, pp. 54–69.

[11] Murthy V.-S. Devarakonda, *File Usage Analysis and Resource Usage Prediction: A Measurement-Based Study*, Ph.D. thesis, University of Illinois at Urbana-Champaign, October 1988.

[12] Murthy V. Devarakonda and Ravishankar K. Iyer, "Predictability of Process Resource Usage: A Measurement-Based Study on UNIX," *IEEE Transactions on Software Engineering*, vol. 15, no. 12, pp. 1579–1586, December 1989.

[13] Kumar K. Goswami, Ravishankar K. Iyer, and Murthy Devarakonda, "Load Sharing Based on Task Resource Prediction," in *Proc. 22nd Annual Hawaii Info. Conf on System Sciences*, January 1989, vol. II, pp. 921–927.

[14] Kumar K. Goswami, Murthy Devarakonda, and Ravishankar K. Iyer, "Prediction-Based Dynamic Load-Sharing Heuristics," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 6, pp. 638–648, June 1993.

[15] J. Baxter and J. H. Patel, "Profiling based task migration," in *Proceedings of the 6th International Parallel Processing Symposium*, Beverly Hills, CA, Mar. 1992, pp. 192–197, IEEE Computer Society Press.

[16] Ian Foster and Carl Kesselman, Eds., *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, 1998.

[17] David Cheriton, "The V Distributed System," *Communications of the ACM*, vol. 31, no. 3, pp. 314–333, March 1988.

[18] F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software - Practice And Experience*, vol. 21, no. 8, pp. 757–786, Aug. 1991.

[19] 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*. June 1996, pp. 38–45, IEEE Computer Society Press.

[20] V.S. Sunderam, "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience*, vol. 2, no. 4, December 1990.

[21] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor – A hunter of idle workstations," in *Proc. 8th Intl. Conf. on Distributed Computing Systems*, San Jose, Calif., June 1988, pp. 104–111.

[22] J. Pruyne and M. Livny, "Parallel processing on dynamic resources with CARMI," *Lecture Notes in Computer Science*, vol. 949, pp. 259–??, 1995.

[23] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky, "Adaptive parallelism and Piranha," *IEEE Computer*, vol. 28, no. 1, pp. 40–49, January 1995.

[24] David Gelernter, Nicholas Carriero, S. Chandran, and Silva Chang, "Parallel programming in Linda," in *International Conference on Parallel Processing*, Aug 1985, pp. 255–263.

[25] Ravi B. Konuru, Steve W. Otto, and Jonathan Walpole, "A migratable user-level process package for PVM," *Journal of Parallel and Distributed Computing*, vol. 40, no. 1, pp. 81–102, Jan. 1997.

[26] R. Namyst and J.-F. Méhaut, "PM^2: Parallel multithreaded machine. A computing environment for distributed architectures," in *Parallel Computing: State-of-the-Art and Perspectives, Proceedings of the Conference ParCo'95, 19-22 September 1995, Ghent, Belgium*, Amsterdam, Feb. 1996, vol. 11 of *Advances in Parallel Computing*, pp. 279–285, Elsevier, North-Holland.

[27] Steven E. Lucco, "Parallel Programming in a Virtual Object Space," in *Proceedings of OOPSLA '87*, October 1987, vol. 22, pp. 26–34.

[28] Rishiyur S. Nikhil, "Parallel Symbolic Computing in Cid," in *Parallel Symbolic Languages and Systems*, October 1995, vol. 1068 of *Lecture Notes in Computer Science*, pp. 217–242.

[29] L.V. Kalé, "The Chare Kernel parallel programming language and system," in *Proceedings of the International Conference on Parallel Processing*, Aug. 1990, vol. II, pp. 17–25.

[30] R. J. Richards, B. Ramkumar, and S. G. Rathnam, "ELMO: Extending (Sequential) Languages with Migratable Objects – Compiler Support," in *Proceedings Fourth International Conference on High-Performance Computing*. December 1997, pp. 180–185, IEE Computer Society Press.

[31] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou, "Cilk: An Efficient Multithreaded Runtime System," in *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, Santa Barbara, California, July 1995, pp. 207–216, MIT.

[32] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," in *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Quebec, Canada, June 1998, vol. 33 of *ACM Sigplan Notices*, pp. 212–223.

[33] Peter Newton, *A Graphical Retargetable Parallel Programming Environment and Its Efficient Implementation*, Ph.D. thesis, University of Texas at Austin, December 1993.

[34] E. Berger and J. C. Browne, "Scalable Load Distribution and Load Balancing for Dynamic Parallel Programs," in *The 1999 Workshop on Cluster-Based Computing, WCBC '99*, Rhodes, Greece, June 1999, pp. 1–5.

[35] Paul Roe and Clemens Szyperski, "Gardens: High Performance Objects, Tasking and Migration for Cluster Computing," Tech. Rep., Queensland University of Technology, Brisbane, Queensland, May 1997.

[36] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A mechanism for integrated communication and computation," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992, pp. 256–267, ACM Press.

[37] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M.E. Zosel, *The High Performance Fortran Handbook*, MIT Press, 1994.

[38] T.J. Sheffler and S. Chatterjee, "An Object-Oriented Approach to Nested Data Parallelism," in *Proceedings of Frontiers '95: The Fifth Symposium on the Frontiers of Massively Parallel Computation*, February 1995, pp. 203–210.

[39] Susan Atlas, Subhankar Banerjee, Julian C. Cummings, Paul J. Hinker, M. Srikant, John V. W. Reynders, and Marydell Tholburn, "POOMA: A high performance distributed simulation environment for scientific applications," in *Supercomputing '95*, 1995.

[40] Jose Nagib Cotrim Arabe, Adam Beguelin, Bruce Lowekamp, Erik Seligman, Mike Starkey, and Peter Stephan, "Dome: Parallel programming in a heterogeneous multi-user environment," Technical Report CS-95-137, Carnegie Mellon University, School of Computer Science, Apr. 1995.

[41] Steven Lucco, "A Dynamic Scheduling Method for Irregular Parallel Programs," in *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation (PLDI)*, San Francisco, June 1992, vol. 27 of *ACM Sigplan Notices*, pp. 200–211.

[42] "Draft Standard for Information Technology—Portable Operating Systems Interface (Posix)," September 1994.

[43] Milind Girkar, Mohammad R. Haghighat, Paul Grey, Hideki Saito, Nicholas J. Stavrakos, and Constantine D. Polychronopoulos, "Illinois-Intel Multithreading Library: Multithreading Support for Intel Architecture Based Multiprocessor Systems," *Intel Technology Journal*, vol. 1st Quarter, 1998, pp. 1–15, 1998, http://developer.intel.com/technology/itj/q11998/articles/art_5.htm.

[44] J. E. Moreira and V. K. Naik, "Dynamic Resource Management on Distributed Systems Using Reconfigurable Applications," *IBM Journal of Research and Development*, vol. 41, no. 3, pp. 303–330, May 1997.

[45] C. Farhat, "A simple and efficient automatic FEM domain decomposer," *Computers and Structures*, vol. 20, no. 5, pp. 579–602, 1988.

[46] George Karypis and Vipin Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," TR 95-035, Computer Science Department, University of Minnesota, Minneapolis, MN 55414, May 1995.

[47] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

[48] A. Gursoy, *Simplified Expression of Message Driven Programs and Quantification of Their Impact on Performance*, Ph.D. thesis, University of Illinois at Urbana-Champaign, June 1994, Also, Technical Report UIUCDCS-R-94-1852.

[49] L. V. Kalé and Attila Gursoy, "Performance benefits of message driven execution," in *Intel Supercomputer User's Group*, St. Louis, October 1993.

[50] Sanjeev Krishnan and L. V. Kale, "A parallel array abstraction for data-driven objects," in *Proceedings of Parallel Object-Oriented Methods and Applications Conference*, February 1996.

[51] Steven E. Lucco and David P. Anderson, "Tarmac: A Language System Substrate Based on Mobile Memory," in *Proceedings. The 10th International Conference on Distributed Computing Systems*, Paris, May 1990, pp. 46–51, IEEE Computer Society Press.

[52] P. H. Geubelle and J. S. Baylor, "Impact-Induced Delamination of Composites: A 2D Simulation," *Composites Part B:Engineering*, vol. 29B, no. 5, pp. 589–602, 1998.

[53] George Karypis and Vipin Kumar, "Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs," in *Proceedings of Supercomputing '96*, Pittsburg, PA, November 1996.

[54] 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," *J. Comp. Phys.*, vol. 151, pp. 283–312, 1999.

[55] A. Grama, A. Gupta, and V. Kumar, "Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures," *IEEE Parallel & Distributed Technology*, vol. 1, no. 3, August 1993.

[56] J. L. Gustafson, "Reevaluating Amdahl's law," *Commun. of the ACM*, vol. 31, 5, pp. 532–533, 1988.

[57] B. R. Brooks and M. Hodoscek, "Parallelization of CHARMM for MIMD machines," *Chemical Design Automation News*, p. 16, July 1992.

[58] J. Vincent and K. M. Merz, "A highly portable parallel implementation of AMBER using the message passsing interface standard," *Journal of Computational Chemistry*, vol. 11, pp. 1420–1427, 1995.

[59] Steve Plimpton and Bruce Hendrickson, "A new parallel method for molecular dynamics simulation of macromolecular systems," *J. Comput. Chem.*, vol. 17, no. 3, pp. 326, 1996.

[60] Y.-S. Hwang, R. Das, J.H. Saltz, M. Hodoscek, and B.R. Brooks, "Parallelizing Molecular Dynamics Programs for Distributed Memory Machines," *IEEE Computational Science & Engineering*, vol. 2, no. 2, pp. 18–29, Summer 1995.

[61] David Brown, Hervé Minoux, and Bernard Maigret, "A domain decomposition parallel processing algorithm for molecular dynamics simulations of systems of arbitrary connectivity," *Computer Physics Communications*, vol. 103, pp. 170–186, 1997.

[62] A. Windemuth, "Advanced algorithms for molecular dynamics simulation: The program PMD," in *Parallel computing in computational chemistry*, T. G. Mattson, Ed., pp. 151–169. ACS Books, Washington, DC, 1995.

[63] Terry W. Clark, Reinhard v. Hanxleden, J. Andrew McCammon, and L. Ridgway Scott, "Parallelizing molecular dynamics using spatial decomposition," in *Proceedings of the Scalable High-Performance Computing Conference*, Los Alamitos, Calif., 1994, pp. 95–102, IEEE Computer Society Press.

[64] Steve J. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, pp. 1–19, March 1995.

[65] Mark Nelson, William Humphrey, Attila Gursoy, Andrew Dalke, Laxmikant Kalé, Robert D. Skeel, and Klaus Schulten, "NAMD— A parallel, object-oriented molecular dynamics program," *J. Supercomputing App.*, 1996.

[66] M. R. S. Pinches, D. J. Tildesley, and W. Smith, "Large scale molecular dynamics on parallel computers using the link-cell algorithm," *Molecular Simulation*, vol. 6, no. 1, pp. 51, 1991.

[67] M. Mitchell Smith, "Histone structure and function," *Curr. Opinion Cell Biol.*, vol. 3, pp. 429–437, 1991.

[68] Helmut Grubmüller, Helmut Heller, and Paul Tavan, "FAMUSAMM: A new algorithm for rapid evaluation of electrostatic interaction in molecular dynamics simulations," *J. Comput. Chem.*, vol. 18, pp. 1729–1749, 1997.

[69] David Brown, Julian H. R. Clarke nad Motoi Okuda, and Takao Yamazaki, "A domain decomposition parallel processing algorithm for molecular dynamics simulations of polymers," *Computer Physics Communications*, vol. 83, pp. 1–13, 1994.

[70] K. Esselink and P. A. J. Hilbers, "Efficient parallel implementation of molecular dynamics on a toroidal network. part ii. multi-particle potentials," *Journal of Computational Physics*, vol. 106, pp. 108–114, 1993.

[71] W. Rankin and J. Board, "A portable distributed implementation of the parallel multipole tree algorithm," *IEEE Symposium on High Performance Distributed Computing*, 1995, [Duke University Technical Report 95-002].

[72] Abdulnour Toukmaji, Daniel Paul, and John A. Board Jr., "Distributed particle-mesh Ewald: A parallel Ewald summation method," in *Proceedings of the Parallel and Distributed Processing Techniques and Applications Conference, Aug 9-11, 1996, Sunnyvale, CA*, 1996.

[73] William F. Humphrey, Andrew Dalke, and Klaus Schulten, "VMD – Visual molecular dynamics," *Journal of Molecular Graphics*, vol. 14, no. 1, pp. 33–38, 1996.

[74] Mark Nelson, William Humphrey, Attila Gursoy, Andrew Dalke, Laxmikant Kalé, Robert Skeel, Klaus Schulten, and Richard Kufrin, "MDScope – a visual computing environment for structural biology," *Comput. Phys. Commun.*, vol. 91, no. 1, 2 and 3, pp. 111–134, 1995.

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

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

[77] A.B. Sinha, *Performance Analysis of Object Based and Message Driven Programs*, Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, January 1995.

[78] Robert K. Brunner Shobana Radhakrishnan and Laxmikant V. Kalé, "Branch and bound based load balancing for parallel applications," Tech. Rep. 99-06, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, May 1999, Submitted for publication.

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

[80] Robert K. Brunner and Laxmikant V. Kalé, "Handling application-induced load imbalance using parallel objects," Tech. Rep. 99-03, Parallel Programming Laboratory, Department of

Computer Science, University of Illinois at Urbana-Champaign, May 1999, Submitted for publication.

[81] Robert Brunner, Laxmikant Kalé, and James Phillips, "Flexibility and interoperability in a parallel molecular dynamics code," in *Object Oriented Methods for Inter-operable Scientific and Engineering Computing*. SIAM, October 1998, pp. 80–89.

[82] Laxmikant Kale, Robert Brunner, James Phillips, and Krishnan Varadarajan, "Application performance of a linux cluster using converse," in *Proceedings of 3rd Workshop on Runtime Systems for Parallel Programming, Rolim et al (Eds)*. IPPS/SPDP, 1999, Lecture Notes in Computer Science 1586, pp. 483–495, Springer.

[83] L. V. Kalé, Milind Bhandarkar, and Robert Brunner, "Load balancing in parallel molecular dynamics," in *Fifth International Symposium on Solving Irregularly Structured Problems in Parallel*, 1998, vol. 1457 of *Lecture Notes in Computer Science*, pp. 251–??

[84] L. V. Kalé, M. Bhandarkar, R. Brunner, N. Krawetz, J. Phillips, and A. Shinozaki, "Namd: A case study in multilingual parallel programming," in *Languages and Compilers for Parallel Computing*, Zhiyuan Li, Pen-Chung Yew, Siddharta Chatterjee, Chua-Huang Huang, P. Sadayappan, and David Sehr, Eds. 1998, number 1366 in Lecture Notes in Computer Science, pp. 367–381, Springer-Verlag.

[85] James C. Phillips, Robert Brunner, Aritomo Shinozaki, Milind Bhandarkar, Neal Krawetz, Laxmikant Kalé, Robert D. Skeel, and Klaus Schulten, "Avoiding algorithmic obfuscation in a message-driven parallel MD code," in *Computational Molecular Dynamics: Challenges, Methods, Ideas*, P. Deuflhard, J. Hermans, B. Leimkuhler, A. Mark, S. Reich, and R. D. Skeel, Eds., vol. 4 of *Lecture Notes in Computational Science and Engineering*, pp. 472–482. Springer-Verlag, November 1998.

VITA

Robert Brunner was born in Cincinnati, Ohio on December 12, 1967. He received his B.S. degree with High Honors in Electrical Engineering from Case Western Reserve University, Cleveland, Ohio in May, 1990. At the University of Illinois he was employed as a research assistant from 1990 to 2000.

He is currently a Research Programmer for the Theoretical Biophysics Group, Beckman Institute at the University of Illinois, Urbana-Champaign. His research interests include parallel computing, molecular dynamics simulation, and web-based collaborative research.