Dynamic load balancing via thread migration

David Cronk
College of William & Mary - Arts & Sciences

DYNAMIC LOAD BALANCING VIA THREAD MIGRATION

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William and Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

by

David Cronk

1999

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of

Of the Requirements for the Degree of

Doctor of Philosophy

[illegible]

Author

Approved, July 1999

William Lee Bynum

William Bynum

[illegible]

Piyush Mehrotra

[illegible]

Phillip Kearns

Virginia Torczon

Virginia Torczon

Sidney H. Lawrence

Sidney Lawrence

Contents

ACKNOWLEDGMENTS

As I look back on the past few years and reflect on what it has taken for me to get to this point in my life, I realize that this journey has been both challenging and rewarding. As I look forward to continued challenges and future rewards I realize that none of this would have been possible without the help and encouragement of numerous people.

I wish to express my deepest gratitude to Piyush Mehrotra for not only being my mentor, but also for being my friend. Piyush not only taught me how to conduct research, but also how to work on my own and to strive for my goals. I have learned many things from Piyush and will remember those lessons forever, and I can only hope that someday I can be fortunate enough to have the opportunity to return the favor by having even half as much of an impact on someone else's life. I would also like to thank David Nicol for giving me a second chance at attending The College of William and Mary and for securing my research position at ICASE. The research environment at ICASE was very stimulating and I had the opportunity to work with many people at the top of their fields. This experience was more valuable than anything I ever learned in a class room.

Bill Bynum was a pleasure to have as an academic advisor and I thank him for his patience, understanding, and willingness to accept me as an advisee in my unusual circumstances. I also thank Drs. Phil Kearns, Virginia Torczon, and Sidney Lawrence for agreeing to sit on my committee.

This work would still not have been possible without the help of many other colleagues and friends. I thank Matt Haines for all the help he offered while we were colleagues and also for his continued support after he left for a position at the University of Wyoming. Bryan Hess and Leon Clancy were tremendous and always willing to help me with any of my technical problems. Kevin Roe was of invaluable service during my time away from ICASE helping answer questions as well as performing tasks that I was unable to perform from a distance. There are too many others to name, but I am thankful for the help and encouragement I received from all the people with whom I worked at ICASE.

I would like to say a special thank you to Jeanette Childress who has been my best friend and has encouraged me through this entire process. She has been by my side for many years and I could not have done this without her. Thank you Jeanette, you are a truly special person.

This work was supported in part by the National Aeronautics and Space Administration under NASA Contract Nos. NAS1-97046 and NAS1-19480, while I was in residence at ICASE, NASA Langley Research Center, Hampton VA, 23681.

List of Tables

List of Figures

ABSTRACT

Light-weight threads are becoming increasingly useful for parallel processing. This is particularly true for threads running in a distributed memory environment. Light-weight threads can be used to support latency hiding techniques, communication and computation overlap, and functional parallelism. Additionally, dynamic migration of light-weight threads supports both data locality and load balancing. Designing a thread migration mechanism presents some unique and interesting challenges. One such challenge is maintaining communication between mobile threads. A potentially more difficult challenge involves maintaining the correctness of pointers within mobile threads. Since traditional pointers have no concept of address space, moving threads from processor to processor has a strong impact on the use of pointers. Options for dealing with pointers include restricting their use, adding a layer of software to support pointers referencing non-local data, and binding data to threads such that referenced data is always local to the thread.

This dissertation presents the design and implementation of Chant, an efficient light-weight threads package which runs in a distributed memory environment. Chant was designed and implemented as a runtime system using MPI-like and Pthreads-like calls. Chant supports point-to-point message passing between threads executing in distributed address spaces. We focus on the use of Chant as a framework to support dynamic load balancing based on thread migration. We explore many of the issues that arise when designing and implementing a thread migration mechanism, as well as the issues that arise when considering the use of thread migration as a means for performing dynamic load balancing. This load balancing framework uses both system state information, including communication history, and user input. One of the basic functionalities of this load balancing framework is the ability of the user to customize the load balancing to fit particular classes of problems. This dissertation provides implementation details as well as discussion and justification of design choices. We go on to show that the overhead associated with our approach is within an acceptable range, and that significant performance gains can be achieved through the use of thread migration as a means of performing dynamic load balancing.

DYNAMIC LOAD BALANCING VIA THREAD MIGRATION

Chapter 1

Introduction

Light-weight threads have become increasingly popular over the past several years. This is particularly true for threads running in distributed memory environments [4, 22, 27]. Uses of threads in distributed environments include, but are not limited to, providing latency tolerance by overlapping communication and computation, and providing support for dynamic load balancing. Dynamic load balancing involves determining the workload on each processor at runtime and transferring work from one processor to another to overcome a load imbalance. In a multi-threaded environment, this movement of work is referred to as *thread migration*, where a thread from one processor is moved, or migrated, to a remote processor. This dissertation describes the design and implementation of Chant, a multi-layered light-weight threads library for distributed memory architectures. Chant was designed and implemented as a runtime system with a user interface using MPI-like (Message Passing Interface) [20] and Pthreads-like [31] calls. The lower layers support point-to-point message passing between threads executing in distributed address spaces and remote procedure calls.

This dissertation focuses on the use of Chant as a framework to support dynamic load balancing based on thread migration. We explore many of the issues that arise when designing and implementing a thread migration mechanism, as well as the issues that arise when considering the use of thread migration as a means for performing dynamic load balancing.

We describe, in detail, the implementation of the migration mechanism, including the method used to transfer the thread state and its associated data. Furthermore, we discuss how we deal with both user-level pointers and system pointers. That is, if data referenced by pointers resides in a different memory locations following a migration than prior to the migration, the pointers referencing the data are no longer valid. We describe a method for tracking pointers and for updating all active pointers following a migration so that all pointers remain valid. Finally, we discuss issues pertaining to communication among threads. Since Chant supports point-to-point communication between threads, we must maintain the location of threads throughout the system, as well as provide a mechanism for forwarding messages that arrive at a processor and are targeted for a thread that no longer resides on said processor.

An important contribution of this work is the use of Chant as a framework for dynamic load balancing based on thread migration. We describe a separate load balancing layer, which has been built on top of Chant. There are several issues we address when discussing this load balancing layer.

One of the first issues that must be addressed when developing any load balancing system is how to estimate the load of both individual processors as well as the entire system. A common method for estimating load in a multi-threaded system is to simply use the length of the run queue. This is not always sufficient, however, so we explore other methods of load estimation.

Another important issue to be considered is the tradeoff between a more balanced execution and increased communication. That is, if balancing the load of a particular execution causes excessive increase in inter-processor communication, the overall execution time may increase. We explore ways to track communication and incorporate this information in any decision making process involving load redistribution.

One of our primary goals in the design of this load balancing layer was the support of a large variety of programming models. Since a particular load balancing algorithm may work well for one class of programs, and may work very poorly for another class of programs, we have attempted to design a generic load balancing layer. We provide an Application Programmer Interface (API) that allows the user to customize the load balancing to fit particular classes of problems. While we provide default load balancing functionality, this API provides the user with the ability to customize the load balancing to varying degrees. We provide the user with the ability to use nearly no customization, to near 100% customization.

Since a primary goal of this work is improved performance of distributed memory parallel computing, it is important that we attempt to quantify the overheads associated with both multi-threading and thread migration. Furthermore, we attempt to demonstrate that performance can be improved through the use of thread migration as a means of performing dynamic load balancing. Therefore, we provide detailed analysis of the overheads associated with our approach, as well as performance results obtained from using our system on a suite of test applications.

The rest of this dissertation is organized as follows: Chapter 2 provides background information on light-weight threads, including the use of threads in a distributed memory environment as well as background on thread migration. It also provides background information on load balancing, including a sample of contemporary load balancing algorithms. Chapter 3 provides a summary of related work. Chapter 4 provides a discussion of the design and implementation of the lower layers of Chant, including the reasons behind some of our design decisions. In Chapter 5 we describe both the design and implementation of the thread migration mechanism used by Chant. This includes how we deal with pointers and how we handle communication among migrant threads. Chapter 6 provides details of the load balancing layer, including an overview of the API. We provide performance results in Chapter 7. This includes performance of the threads system itself, performance of the migration mechanism, and performance of the load balancing layer on a number of test applications. Finally, in Chapter 8 we offer some conclusions as well some ideas on possible directions this research can take in the future.

Chapter 2

Background

In this chapter, we provide some background information that is needed to understand some of the material presented in later chapters. We introduce the concept of light-weight threads including definitions and motivation for using light-weight threads in a distributed memory environment. We follow this with some additional information on thread migration.

Next, this chapter provides some background information on load balancing, once again, including definitions and motivation. We discuss the decision making process, along with a summary of some contemporary load balancing algorithms. Following this is some background information on the use of thread migration as a means of load redistribution.

1 Light-weight Threads

Light-weight threads have been in use for a number of years. However, most of the work that has been done with light-weight threads, has been on uni-processor machines or shared memory multi-processor systems. This section introduces the concept of threads, followed by a discussion of light-weight threads, and why light-weight threads are useful in a distributed memory environment. This is followed by some background information on migration of light-weight threads in distributed memory environments.

1.1 Definitions

A *thread*, as a straightforward concept, is a single independent sequential flow of control. A normal Unix process can generally be thought of as a single thread. Within a thread there is a single point of execution at any instant. Having multiple threads means that at any instant there are multiple points of execution, one for each thread [5].

Threads are generally classified as "heavy-weight", "middle-weight", or "light-weight". A thread's *weight* corresponds to the amount of context associated with the thread. A thread's context consists of its program counter, machine registers, and other control information needed for its execution [43]. A typical Unix process represents a heavy weight thread since its context consists of the entire state of the process. Many contemporary operating system kernels, such as Mach, allow multiple threads within a single address space. This reduces the size of a thread's context since process state information is kept separate. However, the context of the thread and all thread operations are controlled by the kernel and often include more context than the application needs. These kernel-level threads represent middle-weight threads. Exposing all context and thread operations to the user-level allows for a minimal context, and thread operations can avoid crossing the kernel interface. These user-level threads have a much smaller context than kernel-level threads and represent light-weight threads. Due to their smaller context, light-weight threads have a much shorter context switch time (the time it takes to switch control of the processor from one thread to another) than either heavy or middle-weight threads. Unless otherwise specified, the use of the term "thread" in this dissertation refers to light-weight threads running within a Unix process.

Threads that reside in the same process execute within a single address space. This allows different threads to read and write the same memory locations. In particular, the off-stack (global) variables are shared among all threads within the process. Each thread has its own separate call stack and local variables [5].

A thread system uses a Thread Control Block (TCB) for each thread to allow the system to keep track of each thread's state. A TCB is simply a data structure used to store items such as the current stack pointer, the current frame pointer, status of the thread, thread identifier (TID), and other thread specific information. Each thread has a distinct TCB, which is maintained by the underlying threads system.

Threads have traditionally been used in uni-processor and shared memory multi-processor environments. However, they have recently been utilized in distributed memory environments as well [23, 27]. In order to move to a distributed memory environment, we must take into account some additional issues.

1.2 Light-weight Threads in a Distributed Environment

When running a multi-threaded application, it is often necessary to be able to distinguish between different threads. In a uni-processor environment this is trivial, since each thread has a unique thread identifier (TID) associated with it. However, when running in a distributed environment, threads on different processors may share a local TID. This makes it necessary to also maintain a unique global TID associated with each thread. If threads are stationary (never move from one processor to another), this is again trivial, as a tuple consisting of a thread's local TID and its processor id, serves as a unique identifier. However, if thread mobility is to be supported, this is insufficient. In this case, two threads on different processors cannot share a local TID, since there is no guarantee that one of these threads will not move to the processor on which the other resides. For this reason, if thread mobility is to be supported, there must be another way to maintain unique global TIDs. This will be discussed in more detail in Chapter 4.

As stated above, threads running within a single process share an address space, making information exchange relatively simple. However, when threads are running in a distributed memory environment, threads clearly do not share an address space and cannot directly share information with other threads executing in separate address spaces. This makes information exchange much more difficult. While there are several possible solutions to this problem, we employ explicit message passing as a means of sharing data. A common parallel programming strategy is to use MPI, or Message Passing Interface [20], to perform inter-processor communication.

MPI is a standard developed for writing message-passing programs. When we refer to MPI we are referring to the Application Programming Interface (API) rather than a specific MPI implementation. This allows us to ignore implementation details and concentrate on the semantics of the standard. Since we use only communication primitives, rather than the entire standard, we will restrict our discussion to those primitives that relate to our work.

MPI supports point-to-point communication between processes running in a distributed memory environment. The main communication operations we are concerned with are sending and receiving messages. When sending a message, the user must specify a destination process as well as a buffer from which the message must be copied. When receiving a message, the user may specify the process from which the message should be sent, or a wild-card may be used, indicating the message may arrive from any process. Additionally, a buffer into which the message should be received must be specified. Sends and receives are matched up by the use of a message type, or *tag*. That is, a message sent with tag x, will only be retrieved by a corresponding receive that also used the value x in the tag field.

Communication operations can be either blocking or non-blocking. A blocking operation does not return until the resources specified in the call can be reused. When sending a message, this means the message has been copied from the send buffer, and the user is free to reuse this buffer. When receiving a message, this means the message has actually arrived in the specified buffer, and the buffer's contents are available for use. When a blocking operation is called, the entire process is blocked, and does not regain control of the processor until the call has completed.

Non-blocking operations may return before the operation completes, i.e., before the user is free to use resources specified in the call. Non-blocking calls return a *handle*, which may be used for checking the status of the operation. This handle can be used either to wait for the operation to complete (a form of blocking call), or to test if the operation has completed. The resources specified in the call should not be reused until either a wait is called or a test returns true. Once the user returns from a wait call, or tests positive for completion, the user may reuse the resources, including the *handle*, specified in the original non-blocking call.

Threads running in a distributed memory environment can also make use of MPI, though there are some problems associated with its use. One problem is the way to address messages to particular threads. The problem is that MPI only recognizes entities such as processes, but not individual entities within a process (e.g., a thread). Another problem is that MPI blocking calls block the entire process, while we would prefer to block only the calling thread. Both these issues will be addressed in more detail in Chapter 4.

While we have introduced the concept of threads, and have provided some background on threads running in a distributed memory environment, we have not provided any reasons for using threads in such a way. That is, what advantages are there to using threads in a distributed memory environment?

A primary incentive to using threads in a distributed memory environment is the ability to perform latency hiding. This latency can be due to, among other things, message passing or IO calls. If a thread needs data from a remote processor, it can obtain this data through explicit message passing. In this case, the thread may have to wait an unknown amount of time to receive the data. If the thread has no useful work to perform, it can yield the processor to another thread, one that does have useful work to perform. By doing this, the processor remains busy while waiting to receive remote data.

1.3 Thread Migration

Thread migration allows a thread residing on one processor to move to another processor. Figure 2.1 represents a multi-threaded distributed system. In this system, a thread from

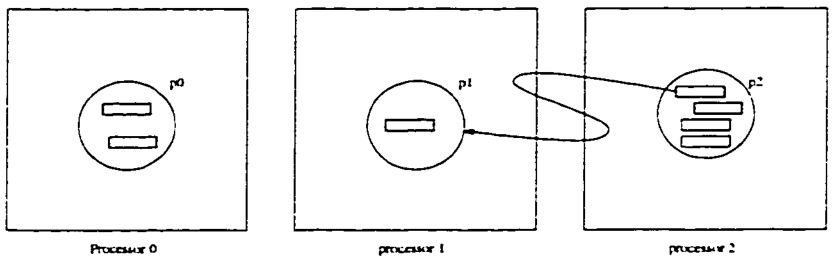

Figure 2.1: *A thread migrates from process p2 to process p1*

process *p2* on processor *2* will migrate to process *p1* on processor *1*. Following the migration, process *p1* will contain two threads, both having access to the same address space. A thread may be migrated for a number of reasons, including improved data locality, improved load balance, and access to system resources that may not exist on all processors.

There are at least two separate models of thread migration. The first model migrates a thread by transferring the data that defines the thread's computation, but little or none of the thread's state [13, 14]. This occurs when threads are migrated before they begin execution, or at very well-defined break-points, when the amount of state is minimal. One example of such a break-point is the end of a main loop of computation. Migration is accomplished by sending the data associated with the thread, and the minimal state information, to a remote processor, where a new thread is created.

A second model, and the one that has been implemented in this work, supports fine-grain load balancing, by allowing a thread to migrate at any point during its execution. In this model, a thread to be migrated is suspended during its execution, and the thread's current state and data are sent to a remote processor. On the remote processor, the migrated thread resumes execution at the point where it was suspended. This migration takes place seamlessly, with the thread having no immediate knowledge that it has moved. This model allows for much better load balancing, because a thread can be migrated at arbitrary suspension points. However, this model requires the ability to migrate the entire state of a thread.

2 Load Balancing

The total execution time of a parallel program is equal to the time it takes for the slowest (most overloaded) processor to complete its computation [13]. It has been shown that in a network of autonomous processors, there is a large probability that, at some point during execution, at least one processor will be idle while there are multiple tasks queued for execution on other processors [17]. In such a situation, it may be advantageous to move some of the work from the busy processors to the idle processors. This movement of work (or load) is referred to as load balancing. For some applications, it is sufficient to balance the load such that every processor has some work at any time. For others, the load must be distributed nearly evenly across all the processors to achieve optimal performance [33].

There are three fundamental issues to be addressed when dealing with a load balancing problem. These issues can be summarized as *when*, *who*, and *which*: *When* is it necessary to perform a load redistribution, *who* (what processors) gets load increases and *who* gets load reduction, and *which* load units are involved in the redistribution. These factors can vary greatly depending on the application in which the load balancing is taking place. Unfortunately, few systems take the application into account when making these load balancing decisions.

There are two distinct types of load balancing policies: static and dynamic. Policies that use only information about the average behavior of the system, ignoring the current state of the system, are referred to as static policies [17]. Static load balancing policies are generally evaluated at compile-time and cannot adapt to unexpected load distributions. Although static load balancing policies work well for regular problems, there exists a large class of problems that have unpredictable computational requirements [49]. These problems are best suited for dynamic load balancing policies. Dynamic policies use the current state of the system to make load balancing decisions at run-time. We focus here on dynamic policies.

The methods of estimating the load of a processor and determining a maintenance policy are of primary importance in designing a dynamic load balancing algorithm. An estimating function can combine several load indicators, including length of the CPU queue, rate of memory occupancy, rate of CPU utilization, rate of communication, and more. However, it has been shown that, in most multi-threaded systems, the length of the ready queue is a good indicator of processor load [37].

The maintenance policy can be based upon the local load situation itself, the local load situation along with the load situation of direct neighbors, or the load situation of any subset of the processors. Additionally, load units (threads in the case of this work) can have a local or global migration space. In an algorithm that uses a local migration space, Work can only migrate to a direct neighbor. When a global migration space is used, work may migrate to any processor in the system [34].

There are a large number of load balancing policies that have been proposed. What follows is a non-exhaustive sampling of these different policies.

- *Dimension Exchange Method* [49]. In this method, any processor that invokes a load balancing operation, exchanges load with each of its direct neighbors successively. The method works by migrating a predefined fraction of excess workload between the two processors. This exchange is done with each neighbor one at a time, without regard to the load on other neighbors. Actions that trigger a load balancing operation may include reaching a load threshold, or expiration of a timer.
- *Diffusion Method* [49]. This method is similar to the dimension exchange method. The only difference is that, in this method, the load of all the neighbor processors is taken into account when deciding the amount of load to exchange between two neighbors.

- *Gradient Model* [34]. In this model, a processor may be in any of three states: L (low), N (normal), or H (high), based on its local load. Additionally, each processor has load and distance information for all other processing elements. Each processor also knows which of its direct neighbors lies on the shortest path to a processor in state L. When a processor enters a state of H, it sends a load unit to its direct neighbor that lies on said shortest path. This load unit is not forwarded along the shortest path, but rather remains on the processor to which it was migrated. However, if this new arrival causes the processor to enter the state H, a load unit will be sent along this shortest path via the same mechanism.

- *Bidding Algorithm* [34]. This algorithm is similar to the gradient model in that it uses the same states (L, N, H). In this algorithm, however, an initiating processor can migrate a load unit to any processor, not only a direct neighbor. In this algorithm, the initiating processor (the one in state H) receives *bids* from processors within some distance *d*. This bid indicates the amount of load the responding processor can accept. The initiating processor then sends a load unit to the processor from which it received the highest bid. The distance value, *d*, changes dynamically, depending upon how many bids are received. If too many bids are received, then *d* is decreased, while if too few bids are received, then *d* is increased.

- *Drafting Algorithm* [34]. This algorithm is similar to the bidding algorithm, except that the under-loaded processor initiates the load balancing operation. In this algorithm, each processor has a table of the loads of all the other processors. When a processor enters state L, it chooses a processor in state H from its local load table, and instructs said processor to migrate a load unit.

- *Centralized* [37]. A centralized load balancing algorithm uses a master-slave model, where the master processor keeps track of the global load state. The master processor broadcasts, to all the slaves, a table of lightly loaded processors at time intervals of *G_DELAY*. *G_DELAY* varies dynamically depending on overall load indications. Slave processors send local load information to the master processor at intervals of *L_DELAY*, which also varies dynamically. When work is to be created on a processor, if said processor is not lightly loaded, then the work is created on a lightly loaded processor, chosen from the table received from the master processor. The target processor is chosen in such a way as to avoid flooding a particular processor.

In terms of computational resources used, these different load balancing policies range from very simple to very complex. However, Eager, et al. [17] contend that the potential benefits of the complex policies do not justify the added complexity nor the added potential for poor results. Their findings show that extremely simple policies that collect a very small amount of state information, and use this information in very simple ways, yield dramatic performance improvement relative to the no load balancing case. Moreover, these simple policies yield performance close to the expected performance of complex policies that attempt to make the "best" choice, based upon the large amount of state data collected. They also show these results to be valid over a wide variety of system parameters. They conclude that simple dynamic load balancing, "is of considerable practical value, and that there is no firm evidence that the potential costs of collecting and using extensive state information are justified by the potential benefits" [17]. This conclusion has not yet been disputed.

3 Load Balancing via Thread Migration

In a multi-threaded distributed environment also, one processor may eventually become overloaded, ending up with a large number of threads while other processors may have very few threads. In Figure 2.1, process *p2* has four active threads, while process *p1* has only one active thread; this represents a load imbalance, assuming all threads do roughly the same amount of work. To attempt to balance this load, a thread from the overloaded process (*p2*) is migrated to the under-loaded process (*p1*), resulting in a more balanced system.

A load balancing policy of choosing the least loaded node for each thread creation is not difficult to implement. However, since threads have varying lifetimes, such a policy may still lead to idle processors when there are more active threads in the system than there are processors. A good load balancing policy should ensure that there are no idle processors as long as there are at least as many active threads as there are processors. Thread migration mechanisms allow the realization of such a dynamic redistribution of threads [38].

The first model for thread migration, which was introduced in Chapter 2 Section 1.3, consists of migrating data rather than the actual thread. Using this model, load balancing can be implemented in a very straightforward manner. The data associated with a thread is viewed as the load unit discussed above. When a load imbalance is detected, data associated with a thread is migrated, using any of the policies discussed above. A new thread is created on the destination node and the computation is continued. The choice of which load unit to migrate can be arbitrary, or it may depend on known communication patterns. In the latter case, a load unit is chosen such that added communication is minimized. Although this model is relatively easy to implement, threads are typically not migrated once they begin execution, or are migrated at specific points in the execution. The result is a very coarse-grain approach to load balancing, often leaving the system unbalanced for considerable periods of time between break-points.

The second model, where a thread can be migrated at any arbitrary point in its execution, offers much more flexibility for implementing a load balancing system. With this model, a load balancing operation can be carried out at any point during the execution of the program. A coarse-grained approach could have the user set points in the code where the system checks the global load and performs any necessary load balancing operations. A finer-grained approach could involve the system checking the global load at either pre-set or varying time intervals. Using varying intervals can allow for the intervals to increase when the system is experiencing little change in overall load, and decrease when the system is experiencing significant load changes. When a load imbalance is detected, an entire thread is viewed as a load unit and is migrated, state and all, according to the load balancing policy in use. This model allows for much finer-grained load balancing, because a thread can be migrated at any arbitrary suspension point. However, this model requires the ability to migrate the entire state of a thread.

In this chapter, we have introduced the concept of a thread as well as discussed the use and functionality of light-weight threads. Additionally, we have discussed the use of threads in a distributed memory environment, as well as the ideas behind the migration of threads across physical processor boundaries.

We have also provided background information on load balancing, as well as some traditional load balancing algorithms. Finally, we have discussed the idea behind using thread migration as a means for load redistribution in a dynamic load balancing system.

The following chapter gives a summary of other systems that provide some of these functionalities. These systems range from simple distributed thread systems, to systems that support thread migration in various capacities.

Chapter 3

Related Work

The use of threads in distributed memory systems has received a good deal of attention over the last several years. We only discuss some of the important systems that provide threads in a distributed memory environment. Discussion of thread systems for single processor, shared memory multi-processors, and distributed shared memory machines can be found elsewhere [5, 40, 50]. Some of the systems we will discuss simply support communication between threads in different address spaces, while others offer thread migration and other advanced functionalities.

1 NEXUS

Nexus [22. 23] is designed as a general-purpose runtime system for task parallel languages. Rather than being a target for end-users. Nexus is intended to be a compiler target. It is currently being used as a compiler target for the languages Fortran M [21] and CC++ [10. 11]. It is designed to run in a heterogeneous as well as a homogeneous environment. The Nexus system consists of nodes. contexts. threads. global pointers. and remote service requests.

A *node* is a physical processing resource. such as a processor in a distributed memory multi-processor. or a shared memory multi-processor. When Nexus starts. an initial set of nodes is created and nodes can be added or deleted dynamically.

Computation takes place within a *context*. where each context relates executable code and one or more data segments to a node. Many contexts can be mapped onto a single node. Once a context has been created on a particular node. it cannot be migrated to another node.

Computation takes place in one or more *threads*. which may be created locally within a context. or within a remote context. Thread routines are modeled after a subset of the POSIX thread specification known as Pthreads [31]. This was done due to the fact that most vendors supply either Pthreads or a similar threads package. This improves the portability of Nexus.

Nexus provides the compiler with a global name space by allowing a global name to be created for any address within a context. This name is called a *global pointer*, and consists of a context identifier and a local address. A global pointer can be moved between contexts, providing the ability to share global references.

A thread can issue a *remote service request* to request that an action be performed on some remote context. This causes the context pointed to by a global pointer to execute a special function, known as a *handler*. The handler is invoked asynchronously by the remote context, which allows two or more handlers to execute concurrently.

2 Panda

Panda [4] is a portable virtual machine designed with the portability requirements of parallel languages in mind. It is currently used to implement the Orca parallel programming system [1].

In order to support portability, Panda was designed using a layered approach. There is a system independent Panda layer on top of a system-dependent layer. This facilitates portability since only the system layer need be modified to port to other architectures.

2.1 Panda Interface

The Panda interface provides Remote Procedure Calls (RPC), totally-ordered group communication, and thread abstractions, with which Panda applications can be built. Totally-ordered Group Communication assures that all members of a group receive all group messages in the same order. The thread interface is based upon the Pthreads [31] and C Threads [15] interfaces.

The RPC interface is based upon the notion of a service that provides a number of operations. A service is implemented by one or more servers that register the services they provide. A client obtains a handle to the server, and sends an RPC request to the particular node on which the server resides. When a request message arrives, a thread is started, which calls the registered function associated with the request.

The group abstraction provided by Panda supports totally-ordered, closed groups. A group of threads being closed implies that only members of the group can send messages to the group. Groups can be created and joined dynamically. If a thread joins a group that does not exist, then the group is created, and the requesting thread is a member of the group.

2.2 System Interface

Panda's system interface hides machine dependencies by providing three abstractions: threads, messages, and communication primitives. Threads are implemented in the system layer, with an interface identical to that of the Panda layer.

The communication primitives include send primitives (uni-cast and multi-cast) and addressing primitives. When the Panda layer is initialized, a message receive handler is registered with the system. All processes run a system layer receive daemon, which handles all incoming messages. When a message arrives, the daemon makes an upcall to the message receive handler. This handler carries out the necessary operations associated with the message.

Messages are stack-like. The sender pushes data fields of specified size and alignment into the message. These fields may include the sequence number and other information needed if the message has been fragmented. The fields are popped in reverse order by the receiver, who then handles the messages appropriately.

3 Mach

The Mach operating system allows for multiple tasks (processes) over multiple nodes (processors). It uses kernel-level threads to support concurrency within a task. A common operation in Mach is the Remote Procedure Call (RPC), where a thread in one task requests some work be done by another task. Recent work has studied the benefits of thread migration as a mechanism for RPC [18, 19]. With this approach, when a thread makes a request for an RPC, rather than a new thread being created in the remote task, the thread itself will migrate to the task, and carry out the request itself. Although a thread may request work be done by a task residing on a different node, the thread migration work is concerned only with local RPC. That is, a thread migrates only when the server task resides on the same node as the client thread.

4 Computation Migration

Computation migration [30] is a technique where a portion of a running thread is migrated to a remote processor for the purpose of remote data access. The idea is based solely on data locality, and can be used as an alternative to either data migration or remote procedure calls. If the data to be accessed is large, then moving the computation to the data can be cheaper than moving the large block of data. This has advantages over RPC-style access, which requires two messages, one for the call, and the second for the reply. These messages can be expensive, since marshaling values into and out of messages can add substantial overhead. However, a major advantage of RPC style access, is that it does not put additional load on the remote processor, other than load absolutely necessary to handle and carry out the request.

Computation migration gives the benefits of both thread migration and RPC. By moving part of the thread's stack to the remote data, locality of access is gained. This can be very advantageous if there is a series of accesses to the same data. The benefits of RPC can also be gained by varying the granularity of the computation migration. This avoids overloading the resources of a single processor, since only the amount of state necessary to improve locality is moved.

Hsieh, Wang, and Weihl [30] implement computation migration by use of compile time transformations. Their implementation generates a special "continuation" procedure to handle the migration. The body of this procedure is the continuation of the migrating procedure. That is, it is the rest of the procedure, following the migration. Client and server stubs are generated to handle the message passing required to invoke the migration. The client stub of the continuation procedure sends a message to the remote procedure to start the server stub. Following migration, if the computation originated from another processor, the thread is killed. This happens if a migrated procedure migrates again, or if a remote procedure call migrates. If the computation originated on the client processor, then the thread waits for the return of the procedure. This allows for a procedure to migrate from one processor to another several times, and return directly to the original processor, rather than having to backtrack the steps. This can save a considerable amount of communication.

Hsieh, Wang, and Weihl implemented computation migration in the Prelude compiler and runtime system [48], and ran experiments on the Proteus simulator [8]. Their experiments show that computation migration outperforms both data migration and RPC in many applications. They conclude that computation migration can be a valuable tool in distributed environments.

5 Emerald

Emerald [6, 7, 32] is a distributed object-based language and runtime system. The primary goal of the designers is to experiment with the use of mobility in distributed programming. The unit of distribution and mobility in Emerald is the object. Some objects contain processes and others contain only data. Emerald offers language support for mobility.

5.1 System Design

Each Emerald object has four components, namely:

- Unique network-wide name.

- Data local to the object: Primitive data and references to other objects.
- Set of operations that can be invoked on the object.
- Optional process.

Objects with processes are referred to as active objects, while objects without an associated process are passive data structures.

Emerald also offers primitives to support mobility. These primitives are as follows:

- *Locate an object.* This primitive returns the node on which the specified object resides.
- *Move an object.* This primitive is simply a suggestion given to the system. The system may choose not to move the object, and additionally, even when an object has been moved explicitly, the system may move the object again on its own.
- *Fix.* This allows the user to *fix* an object to a particular node. Once an object has been fixed to a node, it may not be moved either explicitly or by the system.
- *Unfix.* By unfixing an object you make it a candidate for movement once again.
- *Refix. Refix* is equivalent to a sequence of calls to *unfix, move, fix.*
- *Attach.* This allows a user to *attach* one object to another. When an object is attached to another, if one of the objects moves, then the attached object also moves.

5.2 Implementation

Emerald has been implemented with only one address space per node, where processes are light-weight threads sharing the address space. By having only a single address space per node, objects on the same node can access each other directly. While all objects are defined in the same way, the compiler chooses an appropriate addressing mechanism, storage strategy, and invocation protocol by analyzing the characteristics and use of an object. There are three different styles of object implementation:

- *Global Object.* A *global object* can be moved independently, referenced globally, and invoked by objects not known at compile-time. Global objects are allocated from within the heap. Invocation of a global object may require remote invocation.
- *Local Object.* A *local object* is completely contained within another object. It can only be invoked by its enclosing object and must always move with its enclosing object. This means that all invocations on a local object are local, since the object moves with the only object that may invoke it.
- *Direct Object.* A *direct object* is a local object whose data area is allocated directly in the representation of the enclosing object. Direct objects are primitive types and simple objects.

Each node contains an object descriptor for every global object for which references exist on that node. When the last reference is deleted, the descriptor can be garbage collected. Object descriptors contain state and location information about the object and use direct memory addresses. This means that pointers within an object must be updated if the object is moved. When an object is invoked, its location is first checked, and if it is local it is invoked directly. If an invocation is remote, then the call traps into the kernel where the remote invocation is handled.

The location field in this descriptor is a forwarding address as described in Fowler [24]. This is the last known address of the object, and communication is done with this node for invocation. If the address has been changed, the results will be returned by the new location and the forwarding address updated.

Each global object is assigned a unique *Object Identifier* (OID). Each node has a hash table for mapping OIDs to object descriptors. This mapping is used to locate the descriptor and thus locate the object itself.

The way in which object movement is performed is based on the type of object to be moved. To move a data object, a single message is sent to the destination node. This message includes the data area of the object along with information for re-mapping location dependent addresses. The message also includes OIDs for global object pointers, forwarding addresses, and addresses of object descriptors on the source node.

The destination kernel receives the message and allocates space for the moved object, copies the data into the newly allocated space, and builds a translation table to map original addresses into addresses in the newly allocated space. Object descriptors are located for existing global objects, and new descriptors are created if necessary. The kernel then updates pointers using the translation table.

The literature does not describe the migration of the other types of objects.

6 Amber

Amber [12] is an object based system where the objects are mobile, and the application runs on a network of shared memory multiprocessors. The active objects in Amber are threads. Object operations can be invoked either locally or remotely. Amber is implemented on top of the Topaz operating system for the DEC Firefly. The distribution model and mobility primitives are derived from the Emerald system.

Data placement in Amber is under complete control of the programmer. Threads may be created dynamically. The scheduler supports time slicing, and can be customized to use priority or adaptive policies.

When a thread invokes an operation on a remote object, the thread migrates to the node on which the invoked object resides. Similar to Emerald, objects can be moved, located, attached, and unattached. Additionally, however, objects may be designated as immutable, meaning they will not be changed. In the case of moving an immutable object, the object is replicated on the destination node, rather than moving the object itself. An object may be moved even if it has active invocations. In this case, the thread invoking the object moves along with the object. Object location is under direct control of the programmer. An object is only moved if it is explicitly moved by the user, it is attached to an object that is explicitly moved, or it is a thread object invoking an object that is explicitly moved. Additionally, the system always carries out explicit moves.

Amber uses a type of global virtual memory to facilitate mobility. Dynamic objects are assigned a distinct segment of the global address space, and each object occupies the same virtual address range on any node it visits. The segment of virtual memory occupied by an object on one node is reserved for that object on every other node.

Remote references and locating objects are handled the same way as in Emerald.

Thread migration involves simply copying control information and the thread's stack to the remote node. This is copied to the same address space on the destination node as it occupied on the source node. The descriptors are updated, and the thread put on the ready queue of the destination node. All addresses remain valid since the thread occupies the same memory location.

7 Parallel Adaptive PDEs

Chrisochoides has done work with a multi-threaded dynamic load balancing implementation for parallel adaptive PDE computations [13]. In this work, processors execute many threads, each typically dependent on results of other local or remote threads. Multi-threading is used here for both overlapping communication and computation, and also for overlapping load balancing operations and computation. Load balancing operations include information dissemination, decision making, and data migration. Threads in the system can be in any of five states: *new*, *ready*, *running*, *blocked*, or *dead*.

Problems are broken up into *domains*, *blocks*, *sub-domains*, and *regions* where each computational thread corresponds to a separate region of grid points. Different sub-domains are distributed over the different processors, where a set of contiguous regions makes up a sub-domain. A set of sub-domains makes up a block, where blocks are independent of one another in the whole application. A domain corresponds to the entire application. Each sub-domain has *interface* regions and *interior* regions. Interior regions need only data local to the processor to complete their computation and can execute independently of interior regions on other processors. Interface regions require data from interface regions on other processors and thus must synchronize their execution.

Load balancing is accomplished by migrating threads from overloaded processors to under-loaded processors. This migration is done in such a way as to minimize future communication. The number of grid points within a thread can change during computation in order to achieve better balancing of work loads.

The load balancing approach in this work attempts to ensure that no processor is waiting idle while more than one thread remains to be executed on any other processor. When a processor becomes idle, it requests threads from a subset of processors that are overloaded. This subset consists of neighboring processors only, and work is redistributed via thread migration. Threads are migrated in a way as to minimize overheads due to future message passing. This is done by minimizing the number of grid points that reside on the interfaces of the sub-domains.

Only interface threads are migrated. which causes some interface threads to become interior threads. and some interior threads to become interface threads. Threads are only migrated at the beginning or end of large blocks of computation, and thus true thread migration is not necessary. Thus, instead of thread migration, the system actually performs data migration. During load balancing, a thread that is to be migrated is halted. The data (grid points) associated with this thread is packed into a message and sent to the destination processor, and the thread is killed. On the destination processor, the data is received, unpacked, and stored in appropriate memory locations, and a new thread is created, which performs the remaining computation on the data. The system uses hardware interrupts to ensure that migration requests are handled promptly, minimizing the idle time of the requesting processor.

8 PM^2

PM^2 [38, 42] is part of the ESPACE project (Execution Support for Parallel Applications in high-performance Computing Environments), whose basic functionality is the Light-weight Remote Procedure Call (LRPC). PM^2 has been specialized for data parallel programming and is therefore not a general use package. LRPC is performed by forking a remote thread to execute a specified service. PM^2 is designed on top of a threads package that is a large subset of the Pthreads standard [31], with some additional functionalities. It uses PVM [25] as its communications library. Creations and synchronizations of threads are implicitly managed by the LRPC primitives, leaving the user free from making explicit calls to thread primitives.

LRPC can be synchronous, asynchronous, or asynchronous with deferred waiting. Synchronous calls must wait for a return value, while asynchronous calls continue with computation, not expecting a result. Asynchronous calls with deferred waiting continue with computation, and at some later point, wait for a return value. LRPC calls need arguments for *mode, service identifier, location, priority*, a pointer to arguments for the remote thread, and a pointer to the results (except for asynchronous calls).

Thread migration is the mechanism used for load balancing. Load balancing can be based upon thread priorities, or more traditional load balancing policies can be used. Thread migration is carried out in three steps:

1. The thread to be migrated is frozen, and its descriptor and the useful part of its stack are packed into a buffer.
2. This buffer is sent to the destination processor.
3. The destination processor unpacks the threads descriptor and stack, putting the stack in a newly allocated address space, and the thread is unfrozen, ready to continue its computation at the same point in its execution as when it was frozen.

The current implementation only allows for a thread to migrate itself. Additionally, pointers to heap data are not implicitly maintained. The user is responsible for migrating heap data as well as maintaining the pointers to said heap data.

There are points in a threads execution when the user may know that a thread should not be migrated. For this reason, each thread has a "migratable" state attached to it, which can be set or unset on demand.

8.1 Load Balancing in PM^2

LBMP (Load Balancing with Migration directed by Priorities) is an interesting approach to load balancing used by the PM^2 system. Each thread in PM^2 has a priority associated with it. If thread X has a priority of m, and thread Y has a priority of n, then thread X should have control of the processor m/n as often as thread Y. However, threads are scheduled based on priorities relative to the local processor only. It may be desirable to have priorities that are relevant across processor boundaries. For this to be true, the sum of all thread priorities should be equal on each processor. To compute a distribution of threads that will enforce this condition is an NP-Complete problem [38]. Therefore, a heuristic to determine a "good enough" distribution is used. Whenever the sum of priorities on a processor changes, a load-balancing thread is awakened. If the change is great enough, all other processors are contacted and a "good enough" distribution of threads is determined jointly by all processors. Each processor then carries out the appropriate thread migrations to obtain the new distribution of threads.

Although PM^2 appears similar to the work described in this dissertation, there are important differences. PM^2 is a specialized package, while the work described in this dissertation is targeted for a wide variety of programming models. Additionally, PM^2 supports neither explicit message passing nor the general use of pointers. That is, the user cannot generate messages to be exchanged between threads, and the system does not handle pointers to heap data. On the other hand, the system described in this dissertation provides strong support for point-to-point message passing, and efficiently handles all types of pointers within mobile threads.

Another important difference involves the levels of abstraction offered by the two systems. While PM^2 attempts to abstract the threading concept away from the user, the system described here is intended to be used as a light-weight threads package. That is, PM^2 implicitly handles most of the thread primitives, freeing the user from making explicit calls to thread primitives. On the other hand, Chant attempts to let the user control the multi-threading, and provides a framework within which the user can take advantage of known attributes of the particular application.

9 Xthreads

Xthreads [44, 45] is a light-weight threads library that offers logical concurrency within each processor (via processes and threads), and physical parallelism across processors in a distributed memory environment. More than one process is allowed per processor to offer better performance during blocking system calls. If a thread makes a system call that blocks, the entire process will block, not allowing other runnable threads access to the processor. With multiple processes per process, when a thread causes the entire process to block, the operating system can switch to another process, allowing threads in that process access to the processor.

The major feature of Xthreads, besides a threads library, is the ability to migrate threads across both process and processor boundaries. When a thread migrates, the migrant thread resumes execution at the statement following the point of migration, as if nothing happened. A thread migration is the same thing as a normal context switch, except that the thread resumes execution in a different process.

The migrate function call requires as arguments: a pointer to the thread to be migrated, an identifier for the processor to which the thread is to be migrated, and an identifier to the process within the specified processor to which the thread is to be migrated. The use of a pointer to the thread to be migrated indicates that a thread can migrate itself or migrate another thread. When a thread is to be migrated, the thread's stack and useful state information (program counter and other information) must be sent to the destination process. In order to save both time and space, the thread's stack is used as a message buffer. The useful information is pushed on the top of the thread's stack, and the stack is sent as the message. The destination process receives the message directly into an available stack area, pops the state information from the stack, does the appropriate operations to get the thread in a ready state, and puts it on the ready queue.

Since the stack may not reside in the same memory location on the destination process as on the source, pointers will become invalid. Rather than solve this problem, it is simply suggested that the user avoid the use of pointers.

10 Ariadne

Ariadne [36] is a threads library designed to support process-oriented parallel/distributed simulation. It is designed to run in a Unix environment and focuses on three main goals:

1. *Support for thread migration.* While thread migration can potentially be used for load balancing strategies, Ariadne uses it mainly for remote data access. When a thread tries to access data local to another process, the thread itself is migrated to that process, rather than the data being migrated to the thread or use of a remote procedure call.

2. *Portability and Flexibility.* Portability is supported by the use of Unix libraries rather than machine-dependent calls. This allows Ariadne to be easily ported to other Unix based machines with very little rewriting of code. Flexibility is supported by the use of a customizable scheduler. This allows the user to customize the scheduling policy to fit the needs of a particular application.

3. *Provide a facility for multi-threaded distributed computing.* This is done by providing a clean interface between Ariadne and communications primitives.

Ariadne is a preemptive threads library, meaning it uses time-slicing for scheduling threads within the same process. Threads can be taken off the processor either by the timer expiring (an interrupt), or by explicitly yielding the processor to another thread. Since a thread may be accessing a critical area of its code at the time of an interrupt, the user may explicitly turn interrupts on and off. The default scheduler uses a priority queue with FIFO ordering of execution. However, a customized scheduler can be created via library calls to allow for application specific scheduling policies.

Thread migration in Ariadne is carried out in a way similar to that in Xthreads, with one distinct difference. While Xthreads does not preserve pointers within the threads stack, Ariadne provides users with a primitive for updating stack references. This allows stack pointers to be updated following a migration. However, references to heap data are not preserved following migration.

11 UPVM

UPVM [9, 41] supports multi-threading and transparent migration for PVM applications. A new abstraction, called a User Level Process (ULP), is defined. ULPs are similar to light-weight threads and communicate only through message passing. The difference between ULPs and traditional threads is that ULPs define a private data heap from which all dynamic memory allocations are made. When a ULP is created, space for its data, stack, and heap, is allocated on each processor involved in the computation. This space is allocated in the same memory location on each processor.

UPVM offers thread migration, which allows finer-grained load redistribution than process migration. The thread migration protocol goes through four stages.

1. *Migration Event.* The global scheduler sends a migration message to the processor containing the ULP to be migrated. The process is interrupted and the ULP's state gathered.
2. *Message Flushing.* It is important to preserve messages in any message passing system. UPVM takes the approach of assuring all messages intended for a migrating ULP have been received before allowing the ULP to migrate. This is done by sending a flush message to all other processors and awaiting acknowledgment. Acknowledgment from all other processors indicates that all messages have been received and the ULP is free to migrate. The flush message includes the destination of the migrating ULP for future communication.
3. *State Transfer.* The source processor sends the state, data, stack, and private heap of the migrating ULP to the destination processor. The destination processor places this information in its allotted address region.
4. *Restart.* The destination processor then places the migrated ULP in the appropriate scheduler queue, and the ULP is ready to execute on the new processor.

Since a ULP has the same memory region reserved on each processor, following a migration no address references have changed. This keeps the system from needing to update pointers following a migration. However, this comes at a high cost, since the need for dynamic memory allocation causes scalability problems. Since each processor must allocate space for each thread, the system is limited by the resources of the processor with the fewest resources, rather than the number of processors.

12 Thread Migration with Active Threads

Thread migration with Active Threads [29] supports thread migration between clusters of shared memory multi-processors. This work supports direct access to stack data, but distributed shared memory (DSM) is used for accessing heap data. The implementation is part of the pSather language, where threads are not under user control, but are system-managed, and handled by explicit language constructs.

Since a DSM is used for access to heap data, pointers to heap data do not need any special handling during a migration. Pointers to stack data, however, are handled in much the same way as previously discussed in UPVM [9, 41]. That is, space for each thread's stack is reserved on each cluster, so that the stack of a migrant thread can be stored in the same memory region at the destination as at the source. This causes pointers to remain valid by default.

This redundant memory allocation is done at startup, with the stack space for all possible threads being pre-allocated. This means that all the threads must have the same stack size and, this size must be known prior to execution.

The thread migration mechanism is used for load balancing, and the initial threads are all started on a single cluster. The results are inconclusive since there is no comparison with the work being distributed at startup with no load balancing. While modest speedup is achieved, there is no indication that similar or better speedup could not be achieved by statically dividing the work among the clusters, and using no dynamic load balancing.

In this chapter we have discussed a number of distributed memory light-weight threads packages of various functionality. These packages ranged from simple systems that only support communication to systems that support thread migration and even dynamic load balancing. In the following Chapter we introduce the design of Chant, which is our implementation of a distributed light-weight threads library. It will cover the lower layers of Chant, leaving the discussion of our thread migration and load balancing mechanisms for subsequent chapters.

Chapter 4

Chant

While the previous chapter gave a summary of related work, this chapter gives details about the design and implementation of Chant. Chant is a distributed light-weight threads library that supports point-to-point message passing. It was designed and implemented as a runtime system, with a user interface using MPI-like and Pthreads-like calls.

Chant is designed as a layered system (as shown in Figure 4.1), where the Chant System Interface makes standard communication and thread package calls for efficient communication and thread manipulation. On top of this communication and threads system interface is a layer supporting point-to-point communication. In standard communication packages, such as MPI, there is no concept of any entities besides processes. This means messages can be sent only to processes, not directly to entities such as threads. Chant is designed

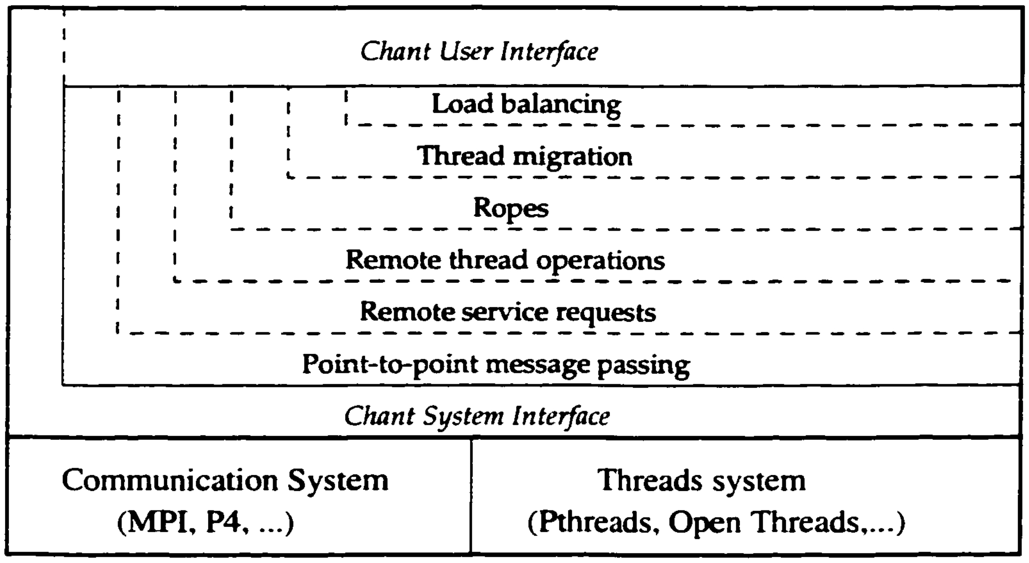

Figure 4.1: *Chant runtime layers and interfaces*

such that a thread in one address space can send a message directly to a thread in another address space, with no intermediate processing or buffering of the message. On top of this layer is the Remote Service Request (RSR) layer, where one processor can instruct another processor to do some work on its behalf. These RSR requests are carried out by an RSR server residing on each processor.

On top of this RSR layer is the layer for remote thread operations. This layer uses the RSR server to perform various remote thread operations, such as remote thread creation. Next there is a layer, called the ropes layer, which supports collective operations and indexed communication among threads.

The next layer is the thread migration layer, which allows threads to be moved across processor boundaries. The final layer is the load balancing layer, which makes use of thread migration to perform dynamic load redistribution.

In this chapter, we discuss the lower layers of Chant, leaving the thread migration and load balancing layers for later chapters. The lower layers of Chant have been implemented on a number of different platforms, including a network of Sun workstations, an Intel Paragon, and an IBM SP2. Additionally, they run on top of several communication libraries. These include MPI (both the LAM and mpich implementations), P4, and NX (the native communication library of the Intel Paragon). Finally, these layers run on top of multiple threads packages, including Pthreads [31], Ports threads, and Open Threads [26].

Chant currently uses the mpich implementation of MPI and Open Threads as its communication and thread libraries, respectively. Open Threads [26] is a threads package developed by Matthew Haines at the University of Wyoming. Open Threads was chosen because it allows the user deep access into the threads system, allowing easy manipulation of the threads and run queues. This allows thread migration to be implemented without making changes to the thread system itself.

Portability is an important issue to be considered. This is why Chant was designed to run on top of existing libraries. This reduces the number of portability issues we must deal with ourselves. While the are certainly architecture dependancies to be address within the communication library, these dependencies are handled by the implementors of MPI. Likewise. there are clearly architecture dependencies involved with porting Open Threads to a new architecture. However. these dependencies are dealt with by the implementors of Open Threads. The only are of Chant where there is any architecture dependence is in the updating of system pointers following a migration of a thread. The details of this will be covered later in this dissertation. Basically there are three requirements for porting Chant to new architectures:

1. There must be an MPI implementation running on the new architecture.
2. There must be an Open Threads implementation running on the new architecture.
3. The manner in which system pointers are handled following a migration must be tailored to the new architecture.

The first two items are managed by the software systems. the message passing and thread implementations. The third item however. simply requires identifying what system values need to be updated and determining their locations. Once they are identified and located it is relatively simple to make the necessary changes.

1 Thread Representation

As discussed in Chapter 2 Section 1.1, most thread systems maintains a Thread Control Block (TCB) for each thread. Our current system runs on top of Open Threads, and thus Open Threads maintains this TCB. We will refer to this as the *Open Threads TCB.*

While the Open Threads TCB contains all the thread specific information important to Open Threads, there exists thread specific information that is required by Chant, but does not pertain to Open Threads. While it would be possible to add the necessary information to the Open Threads TCB, in order to maintain portability, we prefer not to make changes directly to the underlying threads package. Therefore, we introduce a second TCB, which is maintained by Chant. We will refer to this second TCB as the *Chant TCB.*

The Chant TCB is used to store Chant specific information. This includes information needed to support migration, information needed by the load balancing layer, and other information simply needed to maintain the system.

2 Point-to-Point Communication

The design of Chant addresses many of the issues involved in point-to-point communication between threads. Specifically, it identifies the naming of global threads within a process, delivering messages to specific threads, and polling for outstanding messages as primary concerns.

The naming issue is handled through the use of both a global and local thread identifier (TID) associated with each thread. The global TID can be converted to a tuple consisting of a processor identifier and the local TID. This conversion is stored in a statically declared conversion table on each processor. This conversion table simply provides a mapping, between a unique global TID, and a <process identifier, local TID> tuple.

Global TIDs are necessary since Chant threads are mobile, and therefore threads on different processors cannot share the same TID. Consequently, a thread created on one processor must not be assigned the same global TID as a thread existing on a different processor. This is prevented by allowing each processor only a subset of the available global TIDs for assignment to newly created threads. For example, if the maximum number of threads in the system is 100, and there are 2 processors, processor 0 would only assign global TIDs 0-49, while processor 1 would only assign global TIDs 50-99. When a thread migrates from one processor to another, there are no guarantees that the thread's local TID is not in use on the destination processor, thus a thread's local TID may change following a migration. Furthermore, a thread's process identifier changes when it migrates to another processor. Therefore, it is necessary to allow the conversion table to be edited at runtime. Following a migration, the tuple associated with the migrated thread's global TID must be updated to reflect the new values. This update must be performed on each processor in the computation.

While most communications systems support delivery to a particular process within a specified processing element, they do not provide direct support for naming entities within a process. However, in order to support point-to-point communication between threads, a system must have a way to specify the thread for which a message is intended. Chant handles this problem by overloading an existing field of the message header, typically the user-defined tag field. This is done by using half of the field for the TID of the destination thread and the other half for the actual message tag. Although this reduces the number of valid tags from, say, $NTAGS$, to $\sqrt{NTAGS}$, this is a minor restriction since most tag fields are 32-bit integers. This still allows for 65536 unique tags and an equal number of TIDs. An alternative approach would be to put the TID in the body of the message. However, this would be costly, since it would require a message copy at the source processor to insert the TID and another message copy at the destination processor to extract the TID. Additionally, there would need to be an intermediate thread that would receive the message, determine the destination TID, and forward the message to the appropriate thread. The use of an intermediate thread causes this to not be true point-to-point communication and can degrade performance due to the overhead associated with scheduling and executing the intermediate thread.

The Chant interface offers both blocking and non-blocking communication operations. However, when the user makes a call to a blocking receive operation, the system must not use the corresponding blocking call provided by the underlying communication system. This would cause the entire process to block, preventing other threads from gaining access to the processor. Instead, Chant issues a non-blocking communication call, and the system returns a "handle", which can be used to check for the completion of the operation at a later time. If the operation has not completed, then other ready threads can be scheduled, and control returned to the blocked thread only after completion of the non-blocking operation. This approach requires some type of polling mechanism to determine when the non-blocking communication call has been completed. Three basic approaches can be taken. First, the scheduler could check for operation completion, and return control of the processor to the blocked thread only after the operation has completed. However, this requires modification of the scheduler, and most widely available threads packages do not allow such modifications.

A second approach is to have a separate thread check for operation completion, and enable blocked threads to be scheduled following the completion of the operation. However, this creates additional overhead associated with the scheduling and execution of this polling thread. For this reason, Chant employs a third approach. In this approach a thread polls for itself (this is done under wraps, the user has no control nor knowledge of the polling mechanism). When a thread calls a blocking Chant operation, the system calls the corresponding non-blocking MPI operation. The system then checks to see if the operation has completed, and if not, it yields the processor to a ready thread. Once the thread returns to the processor, it again checks if the non-blocking operation has completed. If not, the thread again yields the processor to a ready thread. This cycle continues until the operation completes. Although this approach introduces additional overhead due to unnecessary context switches, we feel it is the best solution available to us at this time.

3 Remote Service Requests and Remote Thread Operations

Remote service requests (RSRs) are used to perform operation on a remote processor. They are different from point-to-point messages in that the destination processor is not expecting the message. RSRs can be used for any operation, but common examples include getting the value from a remote address space (*remote fetch*), executing a function remotely (*remote procedure call*), and processing system requests necessary to keep global state up-to-date (*coherence management*).

Many RSRs require some acknowledgment to be sent back to the requesting thread. It is, therefore, important to handle requests in a timely fashion, to minimize the time the requesting thread is blocked awaiting acknowledgment. However, since MPI does not support interrupt driven messages, Chant was designed to use a polling mechanism by which RSRs can be checked without interrupting a computation thread.

Chant has accomplished this by creating a system controlled thread at initialization called the *server thread*, which is responsible for receiving and handling all RSRs. Using one of the polling techniques outlined earlier, the server thread continually issues non-blocking receive requests for any RSR message. The RSR messages are different from point-to-point messages because they are sent to, and handled by, the server thread, instead of a computation thread. When an RSR message arrives, the server thread gets control of the processor, and handles the request.

When working with threads in a distributed environment, there are certain thread operations that may have global significance. Creating a thread on a remote processor, joining a remote thread (blocking until the joined thread has terminated), and releasing a thread blocked on a join are the only such function currently supported by Chant. Future enhancements might include global mutexes and semaphores along with their associated functions, as well as other functions with global significance. To accommodate global thread operations, Chant uses a processor identifier as a parameter to thread functions with global significance. To carry out global thread operations, Chant uses the RSR mechanism discussed above. For example, to create a thread on a remote processor, Chant sends an RSR to the remote processor, requesting that it create a new thread. The remote processor creates the thread and then returns the global TID to the requesting thread. Other existing global thread operations are carried out in a similar manner, while future enhancements would likewise be carried out in a similar manner. The other currently supported remote thread operations are as follows:

- Joining a thread. Joining a thread causes the calling thread to block until the joined thread terminates.

- Detaching a thread. Detaching a thread allows the data associated with a detached thread to be freed upon its termination.

- Unblocking a thread. Any thread may release, or *unblock*, a thread that has blocked on a condition.

4 Ropes and Data Parallelism

Most current light-weight thread systems do not provide support for *collective operations* and *relative indexing* among threads. Such operations are commonly used in data parallel programs. For example, consider a simple data parallel algorithm for computing the sum over a distributed array (see Figure 4.2). In this example, each thread will compute its local sum, and then participate in a global reduction to obtain the total sum. To execute this example as a set of distributed threads in the midst of other thread activity, and without involving the other threads, a scoping mechanism is needed for identifying the threads that will contribute to the global reduction. *Ropes* [28] provide this mechanism.

The key to collective operations is the ability for the programmer (or compiler) to specify the *scope* of the operation; that is, the entities that will be involved in the operation.

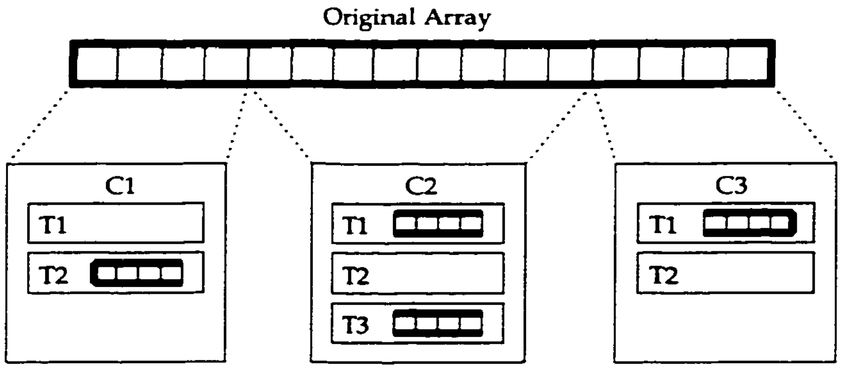

Figure 4.2: *One-dimensional array distributed among four threads in a rope*

Collective operations are typically supported at the process level by the underlying communication system [16], or by standard communication interfaces [20, 46]. For example, MPI [20] provides a mechanism for process scoping called *groups*. However, support for grouping threads within a process is not currently supported by either MPI or existing thread-based runtime systems. Yet, such support is clearly needed if threads are to perform collective operations on a subset of the threads in the system.

Relative indexing allows the programmer to specify spatial relationships among the parallel execution units that express the natural "neighboring" relationships in data parallel programs. Without support for relative indexing among threads, the programmer would be required to assign relative identifiers to the threads. Also, with proper support for mapping processes to processors, relative indexing can also be used to optimize performance by ensuring that an algorithm is correctly mapped onto the underlying topology.

4.1 Requirements

A system for implementing collections among a set of threads (i.e.. ropes) must satisfy the following requirements:

1. The collections are entities whose members can span processor boundaries. and thus their identifiers must be unique within the system.
2. Each collection must keep track of its constituent processors and threads. and operations to add and delete from this list must be performed atomically.
3. Thread ranks within a collection must be unique, so that there exists a one-to-one mapping between the thread identifier with respect to the processor (global thread id), and the thread identifier with respect to the rope (relative index).

4.2 Rope Servers

The requirements listed in the previous section are typically satisfied by having a centralized name server, responsible for allotting rope identifiers and for performing atomic updates to the internal data structures. Distributed algorithms for name servers [39] and atomic operations [35] are well known, but their added overhead and implementation complexity are often unwarranted in an initial design. However, a centralized solution for naming and updating ropes will certainly cause hot-spots. Therefore, our initial design is a two-level approach, derived from the idea of two-level page management schemes for distributed shared memory systems [2], that allows the user to control the contention among servers by dividing the work between two types of servers:

1. a single, global *name server* used to allot identifiers for new ropes, and
2. a *rope server* associated with each rope, which is responsible for all modifications and requests pertaining to that rope.

4.3 Relative Indexing

Spatial relationships play an important role in data parallel algorithms. Most communication systems provide a linear ordering of the participating processors, which allows for *relative indexing* of the processors independent of their actual system address. For example, processes in an MPI group are numbered from *0* to *n-1*. In addition to supporting collective operations, ropes provide a relative ordering for a set of threads that is independent of their actual global address. Thus we say that each thread within a rope is assigned a unique *rank*, starting with zero and increasing linearly. This makes it possible to send a message from thread i to thread $i + 1$ within a rope, without regard to the physical location of the thread. Spatial ordering can also be used to gain performance by exploiting the underlying connectivity of the architecture. However, for this to happen the user must be able to specify a mapping of threads to processes (allowed in Chant) and processes to processors (allowed in MPI).

To support relative indexing, the system must provide a one-to-one mapping between the rank within a rope and the global address of a thread. This is accomplished via a rope translation table to store and retrieve this mapping information. If the translation table is kept in a centralized location, then remote references would be needed for translating all relative indices, which would be prohibitively expensive. Therefore, we replicate this information, and keep a copy of the table on each processor participating in the rope. Figure 4.3 depicts the data structure for the local rope table.

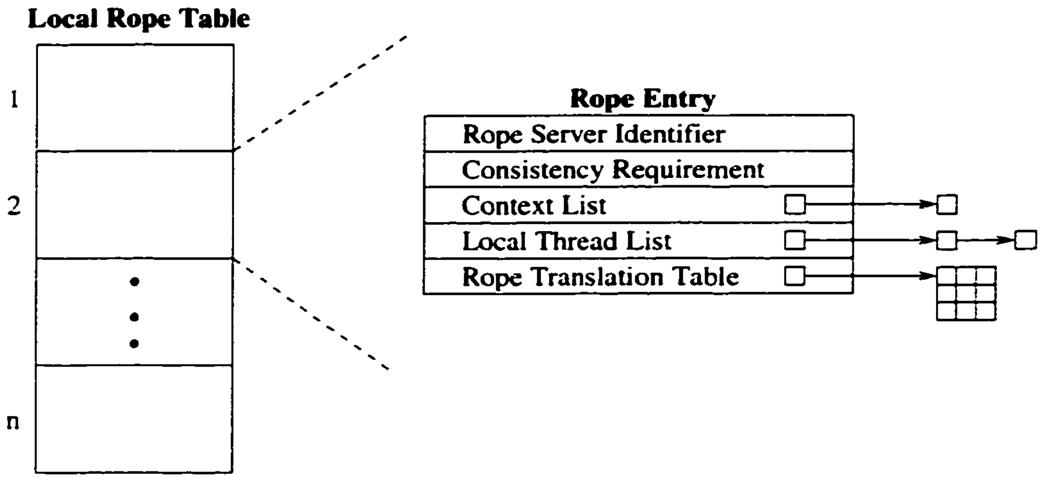

Figure 4.3: *Data structure for local rope table*

Borrowing from earlier work in the area of page coherence for distributed shared memory systems [3], two options were adopted for keeping the distributed translation tables consistent: new information is broadcast so that all tables are kept up-to-date at all times (*strong* *consistency*), or tables are allowed to remain out-of-date until a reference for a thread is generated, causing the information to be retrieved and stored in the local table (*weak consistency*). If each thread in a rope communicates with only a small number of other threads in a rope, then the weak consistency model should result in better performance since the creation cost is so much less. If, on the other hand, each thread in a rope will communicate with many other threads in the rope, the strong consistency model should result in better performance. Determining the crossover point for a given application is an open question depending on the overheads of the two approaches. Therefore, Chant supports both strong and weak consistency on a per-rope basis, by providing an argument to the rope creation routine to specify the consistency requirement. Note that such a system (weak consistency) only works for non-mobile threads.

4.4 Collective Operations

MPI provides the *group* facility for specifying which processes will participate in a collective operation, and ropes extend this idea to the thread level. To do this, each processor participating in a rope must know the other processors in the rope, as well the list of local threads in the rope. This information is maintained for each rope in a rope table (refer to Figure 4.3).

In order to take advantage of system-specific optimizations for collective operations among processors, all collective operations among threads are performed in two steps: at the intra-processor level, and the inter-processor level. For example, consider the *rope_barrier* operation, which performs a barrier synchronization among all threads in a rope. The barrier is performed first among the threads within each processor separately, and then among the processors, as described by the following algorithm:

1. Each thread, upon executing a barrier command, increments a counter monitoring the number of threads within the rope that have reached the barrier. If the counter is not equal to the number of local threads participating in the rope, the thread will block on an appropriate event.
2. If the counter has reached the number of local threads participating in the rope (this information is stored in the rope table, see Figure 4.3), an RSR message is sent to the rope server for this rope. The thread then blocks on the appropriate event.
3. When the rope server has received a message from each processor in the rope, an RSR message is sent to each of the participating processors, informing them that the barrier has been completed.
4. The participating processors receive this RSR message and trigger the events for the local threads, thus completing the barrier.

Ideally, processor-level primitives from MPI, such as *MPI_BARRIER*, should be used to replace steps 2 and 3 in the algorithm. However, the *MPI_BARRIER* is a blocking call, and when invoked by the final calling thread on the processor (step 2), would block the entire processor, including any threads on that processor not related to the rope. This would inhibit one of the key features of a multi-threaded system: the ability to overlap useful computation (in the form of ready, waiting threads) with long-latency, blocking operations. Thus, the implemented algorithm does not use *MPI_BARRIER*, but rather a simple message-combining algorithm that allows other ready threads to execute while the barrier operation proceeds. Should MPI ever support non-blocking collective operations, they would be incorporated into the design as mentioned.

Other collective communication operations, such as reduction functions, can be implemented in a similar two-level fashion.

This chapter has discussed the lower layers of Chant, our distributed memory communicating threads system. We have discussed the design of the point-to-point communication layer, including ways to target specific threads, and ways for polling for messages without blocking the entire processor. Furthermore, we described the RSR layer, discussing the use of a special purpose system thread running on each processor. These layers are integral to the design of both the thread migration and the load balancing layers. Finally, we discussed the concept of ropes, which support collective operations and relative indexing. At this time, load balancing is not supported within ropes, and therefore this layer has no effect on the migration and load balancing layers. However, as ropes are an important part of Chant, and possible future work could concentrate on load balancing within ropes, we felt it was appropriate to introduce this concept here. In the next chapter, we will discuss the design and implementation of the thread migration layer.

Chapter 5

Thread Migration

In a shared memory environment, thread migration is a relatively straightforward concept. This is due to the fact that all memory is directly accessible by the thread, thus data need not be moved. While deciding when to migrate a thread may be more difficult, due to effects such as data locality, the actual mechanism for performing the migration is fairly straight forward. Migrating threads in a distributed environment, however, is a much more difficult task.

In this chapter, we provide the fundamental design of thread migration. We follow this by addressing the problems we encountered when designing the thread migration layer for a distributed memory environment. Furthermore, we present our solutions to these problems and give a detailed description of the implementation.

1 Fundamental Design

Thread migration allows a thread residing on one processor to move to another processor. The basic design of a thread migration mechanism is independent of whether the thread is running in a shared-memory or a distributed-memory environment. The basic migration is performed in three steps:

1. The thread is halted, and its stack, TCB, and any other necessary state information is packed in a buffer.
2. The buffer is sent to the destination processor, and the thread is removed from the source processor's run queue.
3. The destination processor unpacks the thread's state, TCB, and stack, and puts the thread on the run queue.

Following these three steps, the thread is ready to run on the destination processor and no longer exists on the source processor.

2 Issues to be Addressed

As was mentioned in Chapter 5, this work concentrates on the thread migration model that allows a thread to migrate at arbitrary suspension points during its execution. This model presents some difficult problems involving the maintenance of pointers as well as supporting point-to-point communication in the presence of mobile threads.

It is important to point out that, for this implementation, we assume a strict homogeneous SPMD programming environment. That is, the same source code is running on each processor and all processors are identical. This is a very important assumption since supporting thread migration in a heterogeneous environment introduces many additional issues. By assuming homogeneity we need not worry about data and instruction representations, nor do we need to worry about instruction and function locations. This is because global variables and procedures reside in the same location regardless of the processor.

There are several issues involving the use of pointers and support of point-to-point communication that arise in the design of a thread migration mechanism. Maintaining pointers is difficult for a number of reasons. If the pointers refer to data in the thread's stack, then they will only remain defined if the stack is placed in the same memory location on the destination processor as on the source processor. Additionally, if the pointers reference data within the heap, either the heap data must be migrated along with the thread, or there must be a mechanism for a remote data access, since the data will not reside within the new address space.

There are several "solutions" that have been proposed for the pointer problem. Some systems do not allow the use of pointers in migratable threads, while others allow pointers to become undefined following migration [44, 45, 36]. Others still, require the user to maintain pointers to heap data and make users responsible for migrating heap data [38]. All these "solutions" restrict the use of such common data structures as linked lists and trees, and are not considered practical solutions.

Another solution is to perform all memory allocations on all processors, reserving the memory locations for each thread and its associated data in case a thread must migrate [9, 12]. In this approach, when a thread migrates, all its associated data can be stored in the same memory location on the destination processor as on the source processor. This results in severe memory restrictions on the system. Moreover, the number of threads is limited by the memory capacity of a single processor, regardless of the total number of processors.

Some systems use a specialized type of pointer, a "global" pointer [22]. With this method, a "pointer" is a data structure that defines both the processor on which that data resides, and a local pointer to the data that is valid within the specified processor. This requires a more complex data access mechanism in which the owner of the data must first be determined, and if it is not local, a remote data access request must be made. In addition to requiring more remote data accesses than should be necessary, this approach also requires *at least one level of indirection for each local memory access* through a global pointer. This overhead can cause dramatic performance degradation, particularly in a tight inner loop.

This chapter explores both the design and implementation of a more general approach, which allows dynamic thread migration at arbitrary suspension points, direct-access pointers for both heap and stack data, the flexibility to relocate stack and heap data at different addresses on different processors, and continuing to support point-to-point communication between threads. Our design requires keeping track of all dynamically allocated memory in such a way that all the data can be transfered to the destination processor, as well as keeping track of all pointers, so that their values can be updated upon migration, to reflect the new data locations.

Some other issues involve message passing in a system supporting migrant threads. In such a system, there is no guarantee that a thread resides on the processor to which a message is sent. In this case there must be a mechanism for forwarding the message to the appropriate processor. This problem is well studied and a common solution is the one described by Fowler [24]. However, this solution may not be appropriate in a multi-threaded environment. Additionally, a thread may post a receive for a message while residing on one processor, and migrate to another before completing the receive. For this reason, there must be a means for reposting outstanding receive operations following a migration.

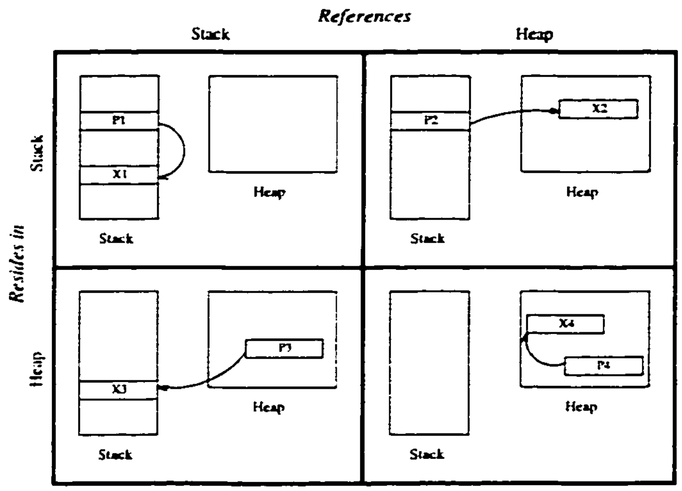

Figure 5.1: *Examples of the four types of pointers*

3 Functional Requirements.

This section outlines the functional requirements of thread migration in Chant. It addresses the different types of pointers that can exist in a multi-threaded system and the ways in which these pointers can be manipulated. Additionally, it addresses issues arising from the support of point-to-point communication between threads. Since threads may move, there must be a mechanism for locating threads, and for handling communication during migration.

3.1 Pointers to Private Data

Private data refers to data that is being referenced by a *single* thread. The pointers themselves may reside in either the thread's stack, or within the heap, and they may reference data located in either the stack or heap. This results in four possible pointer types, as depicted in Figure 5.1:

1. A pointer located in the thread stack and referencing a data item in the stack.
2. A pointer located in the thread stack and referencing a data item in the heap.
3. A pointer located in the heap and referencing a data item in the thread stack.
4. A pointer located in the heap and referencing a data item in the heap.

3.2 Pointers to Shared Data

Sharing data between threads can occur in two ways in terms of pointer use. The first is the use of pointers to global data. Global data refers to data that is declared statically by the process, outside any threads or functions. Since all threads within the process have access to this data, any thread can use a pointer to reference it. These pointers remain valid by default due to the strict homogeneous SPMD assumption. This is because global data resides in the same location on each processor, thus the pointer values remain the same. It is important to point out that this data is global only in a per process sense. That is, it is not accessible to remote processes but has global scope within its own process. A second method of sharing data between threads occurs when pointers from two or more threads reference the same location in the heap.

While the first method for sharing data can be accommodated in a relatively straightforward fashion, the second method is considerably more difficult and is a problem that has not been solved in a satisfactory manner for this work.

3.3 Pointer Manipulation

The ways in which a pointer can be manipulated can be subdivided into three main categories: *assignments*, *updates*, and pointers as *formal arguments*. A general solution to the problem of migrating threads containing pointers should accommodate all three of these categories.

Assignment refers to changing the value of a pointer. This includes assigning to the pointer the address of newly-allocated memory, the value of another pointer, or the address of a variable. Since pointer assignment is such a common operation, it is very important to minimize the overhead associated with these operations. For example, traversing a linked list via an auxiliary pointer performs at least one pointer assignment for each element in the list.

Updates refer to changing the value of the data that the pointer references. It is important that all such updates survive thread migration. That is, it is not sufficient to ensure a pointer references a valid memory location following migration. but that the value referenced by a pointer following migration is identical to the value that was referenced prior to migration.

The integrity of pointers as *formal arguments* must also be protected. For example, consider a thread that is migrated while invoking a function call, and that has at least one pointer as a formal argument. If the function later updates the pointer, the data being referenced must be the same as it was prior to migration.

3.4 Handling Communication

If the thread system supports point-to-point communication between threads, then other considerations must be taken into account. These include tracking the location of the threads. If thread A wants to send a message directly to thread B. then the system must be able to locate thread B. This is necessary so that thread A can send the message directly to the processor on which thread B resides, and thread B can receive the message directly, with no intermediate buffering.

Additionally, a thread may post a receive request, and migrate to another processor before the receive has completed. In this situation, the receive request must be re-posted following the migration. This is due to the fact that the underlying communication library on the destination processor will have no knowledge of the original request, and will be unable to place the message, once it arrives, into the appropriate memory location as specified during the original request.

Finally, a mechanism for forwarding messages must be included. If a message arrives for a thread that has migrated, that message must be forwarded to the new location of the target thread. Additionally, a message may have already arrived, prior to the completion of the receive operation. This would mean the message resides in user space, but the user has no way of detecting this fact following the migration. This message must also be forwarded to the destination processor, so the migrant thread can complete the receive operation. While there has been work done on message forwarding at the process level (e.g. [24]), we know of no work on message forwarding at the thread level.

4 Supporting Pointers

This chapter presents the design of a new thread migration system, which supports pointers. The design revolves around two basic concepts. The **first** of these concepts is the use of a private heap for each thread. This private heap is allocated from the processor's heap as a contiguous block of memory at thread creation time. All dynamic memory allocation performed by the thread occurs within the private heap associated with the thread. This facilitates the calculation of offsets for the purpose of updating pointers following migration, and allows us to migrate all of the thread's heap data in a single message. The **second** concept is keeping track of all valid pointers, which allows all of them to be updated following migration.

This design for thread migration requires the addition of a few new data structures and user-level functions, which we present below.

4.1 Auxiliary Data Structures

To ensure that heap variables reside in a contiguous block of memory that can be easily moved, each thread uses a private heap. This thread heap is allocated from the process heap space, where the size can be specified by the user at thread creation time. For efficiency purposes, this heap is allocated as part of the thread's stack. Essentially, the stack and heap are one and the same, with the only difference being that they grow in opposite directions. This means the stack and heap are part of the same contiguous block of memory.

For the purpose of memory management within a thread's private heap, Chant provides specialized memory allocation and deallocation routines. It maintains of a list of free blocks of memory within the heap and utilizes two fields at the front of each memory block. One field stores the size, in bytes, of the block of memory, while the other stores a flag indicating if the block is free or in use. When a thread attempts to allocate a block of memory from its private heap, the following steps are taken:

1. Adjust the requested size to make sure the block is double word aligned.
2. Adjust the size requested to reserve space for the *size* and *status* fields at the front of the block.
3. Traverse the free list until a block of sufficient size is found:
   - If no block can be found to accommodate the request, merge all adjacent free blocks.
   - Search for a block of sufficient size one more time.
   - If there is still no block available to satisfy the request, return an error.
4. If the size of the block is equal to the requested size, remove the block from the free list.
5. If the block is larger than the requested size:
   - Allocate the requested block size from the end of the free block.
   - Adjust the *size* field of the free block to indicate the remaining size of the block.
6. Set the *status* field of the newly allocated block to indicate it is in use.
7. Return a pointer to the newly allocated block.

Releasing a block is much simpler. The *status* field is set to indicate the block is available and the block is added to the free list.

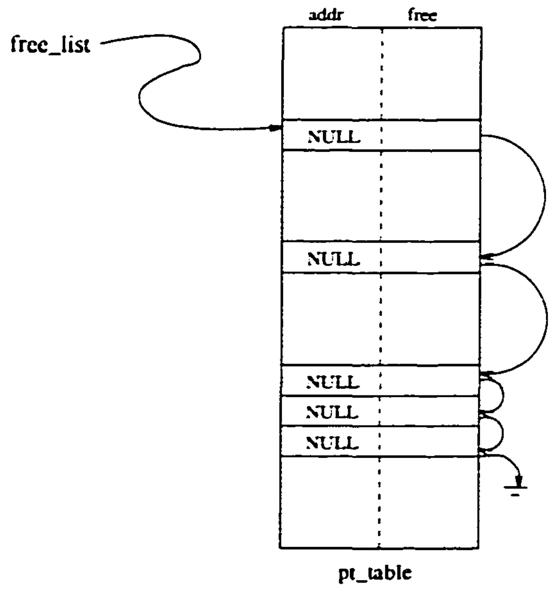

Figure 5.2: *Example of the* **free_list** *for* **pt_table**

These two routines allow for the efficient management of the thread's private heap.

It is also necessary to keep track of all the pointers being used by the thread, and this is done using a table, the *pt_table*, whose entries consist of two fields: **addr** and **free**. The **addr** field is used to hold the address of the pointer itself, and the **free** field is used to facilitate the creation of a free list of table entries. The table itself (depicted in Figure 5.2) is statically allocated at the front of the thread heap during thread creation. Again, the size of the table can be controlled by the user at the time of thread creation.

Finally, the thread control block (*TCB*) of migratable threads must be supplemented to contain an additional four fields: pointer to the base of the thread's private heap, the size of the thread's heap, the size of the thread's **pt_table**, and the head of the free list. These fields are added to the Chant TCB associated with each thread.

4.2 Auxiliary User-level Functions

Our thread migration system relies on the fact that the runtime system knows about all active pointers within a thread. Therefore, a mechanism is needed that allows all active pointers to be registered. Without compiler support we assume that the user will insert the pointer registration calls, though it would not be difficult to modify a compiler to insert the calls automatically.

To register a pointer, a free entry in the **pt_table** must be found, and the **addr** field of this entry set to the address of the pointer being registered. Code for this function is depicted in Figure 5.3(a), where the function **find_free_entry** locates and removes the next element from the free list. The system requires that users register all pointers before they are used, including pointers in dynamically allocated data structures and pointers that are formal parameters to functions.

In conjunction with a registration function, a function is needed to *release* pointers when they are no longer being used, or when they go out of scope. This prevents the **pt_table** table from overflowing and prevents stack memory from becoming corrupted following thread

```
int pt_register(q)                  int pt_release (q)
   void *q;                            void *q;
{                                   {
   int i = find_free_entry();          int i = find_index(q);
   pt_table[i].addr = q;               pt_table[i].addr = NULL;
   return error_code;                  add_to_free_list (i);
}                                      return error_code;
                                    }

          (a)                                 (b)
```

Figure 5.3: *Code for (a) registering and (b) releasing pointers*

```
void* thr_malloc (size)
    int size;
{
   thread_t *t = thread_self();
   void* heap = find_heap(t);
    void* p = find_block (size, heap);
    /* This will include memory managment
         within the thread's private heap */
    return p;
 }
```

Figure 5.4: *Code for thread specific malloc*

migration. When a user releases a pointer, the pointer must be found in the `pt_table` and added to the free list. Code for the release function is depicted in Figure 5.3(b).

While adding an element to the free list is straightforward, finding the element from its address requires a search. A simple linear search will cause on average $n/2$ comparisons per release where n is either the size of the table or the highest index of a valid pointer, whichever is smaller. However, advanced searching and hashing techniques may be employed in the future to reduce this overhead.

Finally, thread-specific **`malloc`** and **`free`** routines for managing memory in the thread's heap are also needed. Code for the former function is given in Figure 5.4.

4.3 Updating Pointers

Migrating a thread takes place from two perspectives, sending data and receiving data. The source processor sends data to the destination processor, which processes this data. Figure 5.5 gives an outline of the steps taken by each.

Source Process	Destination Process
Send size of thread's stack	Receive size of thread's stack
Send Open Threads TCB	Receive Open Threads TCB
Send Chant TCB	Receive Chant TCB
Send Thread's stack	Receive thread's stack
Receive thread's new local thread id	Send new local thread id to source
Inform other processors of new location of thread	Update pointers
Forward messages intended for thread	Repost outstanding receives

Figure 5.5: *Steps taken to migrate a thread*

The source processor first sends a message informing the destination processor that a migrant thread is about to arrive. This message contains the size of the migrant thread's stack. The source processor then sends the Open Threads TCB followed by the Chant TCB. To complete the actual migration, the source processor sends the migrant thread's stack to the destination processor. Thus, the source processor sends four messages to the destination processor to migrate a thread. After the thread has been migrated, the source processor receives, from the destination processor, the new local thread *id* of the migrant thread. This new thread *id* is sent to all the processors allowing each processor to update the thread's location for future communication. Finally, the source processor leaves a stub of the migrant thread for the purpose of forwarding messages. This message forwarding mechanism will be explained in Chapter 5.2.

The destination processor receives a message letting it know a thread is about to arrive, including the size of said thread's stack. The destination processor then allocates memory for the arriving thread's stack, its underlying thread package's TCB, and its Chant TCB. Next the two TCBs are received and the fields updated as needed (some fields have processor specific data that must be updated to reflect the new processor on which the thread resides). Next the thread's stack is received from the source processor and stored in the newly allocated memory. Some bookkeeping is required to handle pointers and communication. This is described in more detail below. Finally, the thread is added to the run queue of the destination processor. When it gets control of the processor, the thread will resume by executing the same instruction it would have executed had there been no migration, making the migration seamless.

Following a migration, the data referenced by pointers will most likely reside in different memory locations. This means that, after a migration, the values of pointers are no longer valid. Consequently, the value of both user-level pointers and system pointers must be updated on the destination processor if they are to remain valid.

User Level Pointers

Following a migration, the migrant thread's **`pt_table`** is available to the system. To update the **`pt_table`**, each valid entry in the **`pt_table`** (**`addr`** != NULL) is examined. By knowing both the base of the old stack and heap (available from the thread control block) as well as the base of the new stack and heap, the **`addr`** field of the **`pt_table`** entry can be used to calculate the new address of each pointer. These new addresses are then placed in the **`addr`** field of the table.

The values stored in the pointers themselves also need to be updated, since the location of the data that they reference has now changed. Since the new address of the pointer has been computed, the pointer can be accessed directly and its current value, i.e., the value on the old processor, can be examined. The following scenarios are possible:

- If the pointer is NULL, then it is not referencing anything and should be left alone.
- If the pointer references global data, then nothing is done since it is assumed that the global data is located at the same addresses on all processors.
- If the pointer references data in the stack or heap, then the same address computation that was done for the pointer addresses is applied, and the value of the pointer updated.

An example of this calculation is shown next. Consider a thread with a pointer residing on the stack, which references data residing in the heap. Let the thread migrate from processor A to processor B, with the following pertinent values:

Processor A	Processor B
• heap base: 1000	• heap base: 1500
• stack base: 200	• stack base: 550
• pointer address: 226	• pointer address: 576
• pointer value: 1013	• new pointer value: 1513

The pointer has an offset of 26 (226-200) from the base of the stack. This translates to an address of 576 (550 + 26) on processor B. The system now accesses memory location 576 on processor B and retrieves the value 1013. This is an offset of 13 (1013-1000) from the old base of the heap, and translates to a new value of 1513 (1500 + 13) on processor B. The system now updates memory location 576 with the value 1513, and the pointer in question now references the correct data.

A final note on user-level pointers: it is essential that the value of a pointer is checked before the *offset* is applied. This is due to things such as linked lists, which use NULL as a terminating pointer. Updating this final pointer from NULL will result in a non-terminated linked list and will eventually lead to undefined results in the user-level code.

System Pointers

When we use the term *system pointers*, we are referring to three different sets of pointers: pointers used by Open Threads, pointers used by Chant, and machine dependent pointers.

Open Threads has, within its TCB, only two non-machine dependent pointers that must be updated. The first is the location of the thread's stack. This field is filled in with the new location of the thread's stack following its allocation. The second is used for queue management. This pointer is automatically updated properly when the migrant thread is put back onto the run queue following migration.

The Chant TCB has only one pointer that must be updated following a migration. This is the pointer to the Open Threads TCB. This field is simply filled in following the allocation of the Open Threads TCB. This is the only field that requires updating because all the lists used by Chant are stored as arrays within the Chant TCB, and are thus automatically filled in when the TCB is received.

This work has been implemented in a strict homogeneous SPMD environment, running on a network of Sun Sparc workstations. In this environment, most of the machine dependent system pointers are maintained by default. This is because instruction addresses are the same regardless of process, i.e., pointers such as return addresses remain valid following a migration, while other system pointers are actually offsets into the stack, and hence are not affected by the migration. Two exceptions. however, are the stack pointer and frame pointers. During a context switch, the current stack pointer is stored in the Open Threads TCB. Following a migration this stack pointer is updated the same way as user-level pointers are updated. by applying the aforementioned *offset* to the old value.

Upon a procedure call. a new stack frame is created at the top of the stack. Within this stack frame, there is a frame pointer, which is a pointer to the previous stack frame. This frame pointer is located at a machine dependent offset from the current stack frame. This frame pointer is the address of the previous stack frame, and not an offset within the stack. Clearly this must be updated using the same *offset* as discussed above. However, each stack frame has a frame pointer, referencing the previous stack frame. which is the stack frame of the calling procedure. Therefore the entire stack must be traversed, applying the *offset* to each frame pointer, and then using the new value to access the previous stack frame. This continues until the frame pointer is NULL, indicating the first stack frame has been reached.

At this point, all the pointers have been updated and the thread can be put on the run queue of the destination processor.

This implementation is specific to the Sparc architecture, and porting Chant to another architecture could possibly require significant changes to the current implementation. However, these changes should only affect the maintenance of system pointers. The solution we have described for user-level pointers is architecture independent.

5 Handling Communication

In our system, a thread can send a message directly to another thread residing in a different address space. This raises some issues if the threads are allowed to migrate. First, the system must globally maintain where each thread is currently residing. Second, messages that arrive for a thread, after the thread has migrated elsewhere, must be forwarded. Finally, if threads may migrate while receives are pending, these receives must be reposted on the destination processor.

5.1 Maintaining the Location of Specific Threads

Since we want to allow point-to-point communication between threads, it is important that each processor maintains a relatively up to date table specifying where each thread resides. This is done by assigning a global thread id (*TID*) to each thread upon its creation. Each processor keeps a translation table, which is used to convert a global *TID* to a tuple consisting of <*processor, local TID*>, which in turn is used for communication.

After a thread migrates to another processor, the source processor waits for a message from the destination processor informing it of the new local *TID* of the migrant thread. The source processor then uses the RSR layer to broadcast the new <*processor, local TID*> tuple, along with the thread's global *TID*, to every other processor in the computation. Each processor then in turn updates its translation table, allowing messages to be directly sent to the thread on the new processor, rather than being forwarded by the old processor. Each processor also responds to the source processor with information needed to forward any messages sent to the migrated thread on the source processor, but not received.

This approach does add non-negligible overhead to the migration operation and raises questions of scalability. Since one message is broadcast, and another gathered from every processor, as the number of processors increase so will the overhead. However, for a single migration, this overhead remains low, even for a large number of processors. For this reason, this approach should scale reasonably well if the number of thread migrations is relatively low. However, if an application uses frequent migration, this approach will probably prove inadequate when running on a large number of processors. Thus, this approach needs to be studied further if large numbers of processors are to be used.

5.2 Forwarding Messages

Let's consider a computation in three processors, *P0*, *P1*, and *P2*, in which thread *X* has just migrated from *P1* to *P2*. The system must handle all unprocessed messages sent from

P0 to *P1* that were intended for thread *X*. If we wait to deal with forwarding messages until after *P0* has been informed of thread *X*'s new location, we can assume that all messages that need to be forwarded have already arrived at processor *P1*.

With this assumption, there are two scenarios to be considered. The first scenario is as follows: thread *X* posted an asynchronous receive prior to migration, the message arrived and is in user space on *P1*, but the thread has not checked for completion of the receive. Thread *X* has no way of knowing the message had arrived once it has migrated to *P2*, and thus the message residing in user space on *P1* must be forwarded to thread *X* on *P2*.

The second scenario is as follows: A message has been sent from *P0*, intended for thread *X*, but thread *X* has not posted a receive for this message. In this situation the message is stored in a system buffer waiting for a matching receive. This message must also be forwarded to thread *X* on *P2*.

As has been explained previously, in a multi-threaded environment it makes little sense to use synchronous receive calls, since these will block the entire processor. Chant, therefore, forces all receives to be asynchronous, and returns a *handle* with which to check for completion. In our system, each processor keeps track, on a per processor basis, of the number of messages that have been sent to each thread in the computation. That is, processor *P0* knows how many messages have been sent from *P0* to *P1* intended for thread *X*. Additionally, each thread maintains a list of unprocessed receives. When a receive is posted. the posting thread stores in this list the *tag*, *handle*, message size, and address of the buffer into which the message is to be received. Each thread also keeps track of the number of messages that have been processed from each processor. We first explore our message forwarding system below ignoring the issue of message tags.

After it has updated its translation table with thread *X*'s new location, processor *P0* sends a message to *P1*, indicating the number of messages intended for thread *X* that have been sent from *P0* to *P1*. At the same time, thread *X* has kept track of the number of messages it has processed from *P0*. The difference between these two numbers represents the number of messages that must be forwarded to *P2*. When thread *X* migrates to *P2*, a *stub* remains on *P1* for the purpose of forwarding messages. This stub has access to all the private data the system maintained for thread *X*.

This leaves us with two possible situations on processor *P1*. First, there may be more messages that need to be forwarded than unprocessed receives. In this situation, we know that all posted but unprocessed receives must have completed, and the messages are in thread *X*'s user space. The difference between the number of messages sent and the number of receives posted represents the number of messages that are in the system buffer waiting for matching receives. The *stub* on *P1* uses the *handles* stored in thread *X*'s list of unprocessed receives to complete each receive, and forwards the buffers (the address is stored in this list) to *P2*. Then, for the messages that have not been received, the *stub* simply posts receives to retrieve the messages from the system buffer, and forwards the messages to *P2*.

The second situation is when there are more unprocessed receives than messages that need to be forwarded. This indicates that all the messages that need to be forwarded have been received by receive calls, and reside in thread *X*'s user space. However, we do not know which receive calls have completed and which have not. In this situation, the ***stub*** on *P1* simply checks each unprocessed receive for completion, using the ***handle*** stored in thread *X*'s list. If the message has been received, the ***stub*** forwards the buffer to *P2*. A non-completed receive means the message was not sent prior to *P0* updating its translation table, and therefore the message will be sent directly to *P2*.

The system as described above can easily be extended to check for messages with all possible *tags*. It can also be easily extended to account for messages from each processor in the computation.

There is, however, one additional issue that arises with this approach. As stated earlier, a receive call can either specify the processor from which the message is supposed to originate, or it can use a wild-card, receiving a message from any processor. Receives using a wild-card cause a slight alteration in the message forwarding process described above. However, receives specifying the processor cause greater problems. Suppose in the above example, thread *X*, after it migrates to *P2*, posts a receive specifying it wants to receive a message from *P0*. Suppose also that the message is sent from *P0* before said migration and must be forwarded to *P2*. If the message is forwarded, the underlying communication system thinks the message originated from *P1*, and the receive operation will never complete.

Therefore, simply forwarding the messages will not work. To overcome this problem, we bounce all messages back to the sender rather then forward them directly. That is, when a message from *P0* is received by the *stub* on *P1*, and needs to be forwarded, it is sent back to the RSR server on *P0* as part of a remote service request. When the RSR server on *P0* processes the remote service request, it knows the new location of thread *X*, so it simply resends the message to thread *X* on processor *P2*. This way thread *X* receives the message from *P0*, as expected. This sequence of events is pictorially depicted in Figure 5.6.

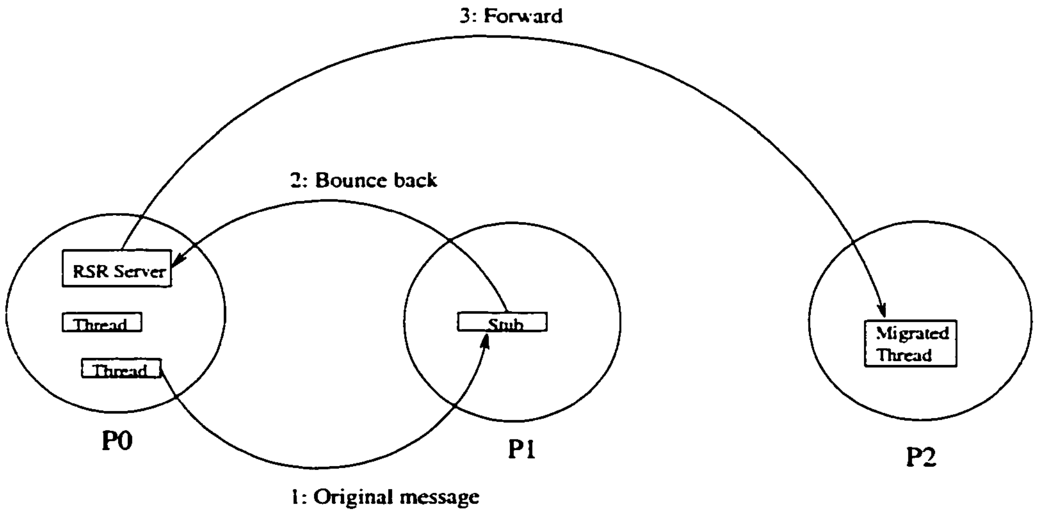

Figure 5.6: *Bouncing a message off the sender to the new location*

The primary drawback to this system is that it requires three message transmissions, while only two messages transmissions would be required if the message could be forwarded directly. Additionally, the use of the RSR server adds overhead. The alternative is to prohibit the user from specifying from which processor a message should arrive. However, this limits the functionality of the message passing system and would be an unreasonable restriction.

5.3 Reposting Receives

If a thread posts a receive and then migrates to another processor, the underlying communication system on the destination processor has no knowledge of the earlier receive call. This would result in the receive never completing. Additionally, the call cannot even be checked for completion, since the returned *handle* is a pointer to a structure on the old processor, and the thread has no access to this structure following migration. Therefore, all outstanding receive calls must be reposted following a migration.

This is done by keeping track of all receive calls in a table. This table includes all the parameters used in the receive call, including the buffer into which the message is to be received. It also includes a *status* field, which is set to PENDING when the receive is posted and to FREE when the message is processed. This allows easy traversal of the table, checking for pending receives, which must be reposted.

Following the migration, the address of the buffer must be updated in the same way as other pointers are updated (see Chapter 4.3). Additionally, all the parameters must be extracted from the table, and are used to repost the receive. Following the reposting, the new *handle* is stored in the table, so subsequent checks for completion use the correct handle.

This chapter has addressed the design and implementation of the thread migration layer of Chant. We have discussed issues involving both the use of pointers and the support of point-to-point communication. We have also supplied various proposed solutions to the pointer problem, and described our solution to the problem. Finally, we gave a detailed description of our implementation of the thread migration mechanism. This thread migration layer is now used to develop a new layer on top of Chant. This layer provides a generic framework within which a user can develop a customized load balancing system, tailored to specific applications.

Chapter 6

Load Balancing

This chapter uses the thread migration mechanism discussed in Chapter 5 to describe the design and implementation of a load balancing layer. This load balancing layer is intended to work as a framework within which users can make use of thread migration, and a supplied Applications Programmer Interface (API), to customize load balancing to fit their particular needs. We revisit some of the issues involved in the design of any load balancing system, as well as introduce some additional concerns that arise in a multi-threaded approach. The primary goal of the load balancing layer is to offer the user control over the decision making process, while at the same time allowing the user to ignore unnecessary details.

The primary phase of load balancing is the decision making phase. In this phase the system must decide: Is a load redistribution necessary? If yes, what processors should be involved in the load redistribution? If a processor should send work to another processor, which load units (threads) should be migrated? Unfortunately, there are no general answers to these questions since different applications perform better under different load balancing algorithms. That is, a load balancing algorithm that performs well for one application, may perform poorly for another application. There are many factors that can affect this decision making phase. For this reason, our primary goal while designing the load balancing layer was to provide support for building customized load balancing systems tailored to specific applications. This has been accomplished by providing a default load balancing implementation in which the user can easily override the default choices. The level of customization is left to the user, from nearly no user support, to a near 100% customized implementation.

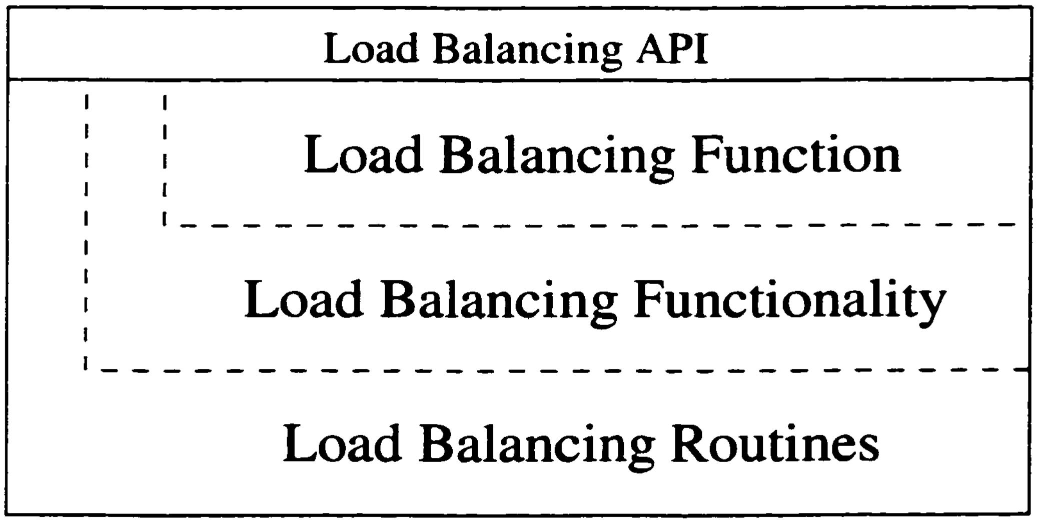

Figure 6.1: *Load Balancing sub-layers*

The load balancing layer is itself built using a layered approach, consisting of three sub-layers. The sub-layers in Figure 6.1 would fit into the *Load balancing* layer of Figure 4.1 with the *Load Balancing API* being part of the Chant User Interface from Figure 4.1. The bottom sub-layer consists of lower level load balancing routines. These routines range from gathering state information to manipulating the run queue. The middle layer provides load balancing functionality. These routines either determine which threads to move where, or instruct the system to move specific threads to specific processors. The routines that are used for decision making can be customized by the user, while the routines that instruct the system to move threads are, by their nature, statically implemented. These routines make use of the lower level routines to make decisions and to move threads. The top-most sub-layer is the actual load balancing function. This is the function that is called by the system whenever it is to attempt load balancing, i.e., enter the decision making phase. We provide a default load balancing function, but allow the user to register a customized load balancing function that may make use of the two lower layers.

The customizability of the top two layers allows the decision making phase to range from very simple to extremely complex. If the decision making phase is very simple, it may be advantageous to attempt load balancing often. However, if the decision making phase is extremely complex, the system may only wish to attempt load balancing when there is a good likelihood that a load redistribution is necessary. This is because an extremely complex decision making phase may take a long time to complete. If this phase is entered often with no redistribution necessary, the system is spending a lot of time in this phase that could be better spent on the actual computation. For this reason, it is important to give the user control over how often this phase is entered.

Our load balancing mechanism extends the use of the *RSR server*, discussed in Chapter 4, Section 3, by adapting it to carry out load balancing operations. When users enable load balancing, they specify how often the *RSR server* should attempt to perform load balancing. Since the situation where load balancing is not necessary often indicates a fairly stable system, this frequency may vary. The user can specify that the *RSR server* never attempts load balancing (on a processor by processor basis), or attempts load balancing every time it gains control of the processor. Alternatively, this frequency can vary linearly or exponentially. A linear variation means that load balancing is attempted every *nth* time the *RSR server* gets on the processor, where n starts at 1, and increases by one each time a load redistribution is not necessary. The parameter n is reset to 1 following an actual load redistribution. An exponential variation means that load balancing is attempted every *nth* time the *RSR server* gets on the processor, where n starts at 1, and doubles each time a load redistribution is not necessary. The parameter n is reset to 1 following an actual load redistribution.

When the *RSR server* gets control of the processor, it checks to see when load balancing was last performed. If it has been long enough (using the frequency discussed above), the *RSR server* makes a call to the load balancing function. If load balancing is actually performed, it resets the frequency counter. If no load balancing is performed, the *RSR* *server* uses the user defined frequency to indicate when load balancing should be attempted again.

This functionality does not affect the decision making phase itself, rather it simply affects how often the decision making phase is entered. There remain several factors that affect the decision making phase itself. These include, but are not limited to, communication histories, how the load is estimated, and which processors are involved in the decision making. These factors, and others, are described in more detail below.

The rest of this chapter discusses the sub-layers of the load balancing layer in more detail. This discussion includes descriptions of default implementations, as well as descriptions of the information returned by the various routines provided through the load balancing API. Furthermore, we provide brief discussion on how this information may be used to customize the load balancing system.

1 Lower Level Load Balancing Routines

The lower level load balancing routines are used to set system parameters, manipulate the system, and obtain both system and thread characteristics. This sub-layer is accessed by both the user and the upper sub-layers of the load balancing layer. The characteristics set by these routines are used by the system and the decision making phase, while the values returned by some of these routines can be used to customize the decision making phase.

1.1 Set System Parameters

One of the most important routines in this sub-layer is the one that allows load balancing to be enabled and disabled dynamically. This may be important for a computation that goes through phases, some of which require load balancing, and some of which do not. For example, consider a rendering algorithm that renders multiple images. In many such applications, the rendering stage is the only one that requires load balancing. In such a situation, the user may enable load balancing prior to rendering an image, disable it once the image has been rendered, and re-enable load balancing when the next image is ready to be rendered.

When load balancing is enabled, the user must supply three parameters. These are upper and lower thresholds and a frequency for attempting load balancing. The upper and lower thresholds are used by the default load balancing function to determine if a load redistribution is needed. These threshold values are also available to any user-supplied load balancing function. The default load balancing function will be discussed later in this chapter. The frequency, as described above, is used by the system to determine how often the decision making phase should be entered.

If a computation is running on a large number of processors, gathering information from each processor in the computation may introduce an excessive amount of overhead. In this situation, it may be advantageous to only include a small sub-set of the processors in the decision making phase. A common use of this concept is to only use immediate neighbors of the processor initiating the load balancing operations. These neighbors can be physical neighbors, as with a mesh architecture, or virtual neighbors in an environment such as a network of work stations. This is the way the *Dimension Exchange* and *Diffusion* [49] methods work as described in Chapter 2.

This sub-layer provides a routine that sets system parameters to define the load balancing domain, i.e., the set of processors to be involved in load balancing operations. This domain can include all the processors within the computation, or any sub-set of processors. This routine is called on each processor in the computation, and sets up processor specific domains. Different processors may be included in different load balancing domains. This domain may be used for any subsequent global operation.

1.2 Thread Characteristics

The routines discussed in the previous section neither affect the execution of the application nor do they gather any information regarding the system. They simply set parameters that will be used to decide when the decision making phase should be entered and which processors should be included in the decision making phase.

The routines discussed in this section are used to dynamically alter thread characteristics as well as obtain information about specific threads. These routines work strictly on a local basis. That is, they deal with characteristics of threads residing on the processor from which the routines are called. The routines that dynamically alter a thread's characteristics are strictly user-level routines. That is, they are called by the application program, and never from the system itself. The routines that gather information, on the other hand, are used by the decision making routines, which will be discussed in Chapter 6, Section 2. Default implementations of these decision making routines are provided, though the user has the option of providing customized routines.

One important characteristic of a thread is its migratability. The user may create certain special threads that should remain on the same processor throughout their lifetimes. Chant supports this by allowing the user to specify, at thread creation time, if a thread is migratable or non-migratable. A non-migratable thread will never migrate to another processor. A migratable thread, by default, can be migrated either by the system, or explicitly by the user. However, there may also be a situation where a thread should be allowed to migrate at certain times, and not be allowed to migrate at other times. For this reason, a thread that is created as migratable, may be set to user-migratable, which means it can only be explicitly migrated by the user, but not by the system. A migratable thread can be dynamically switched between being system migratable or strictly user-migratable. Additionally, we provide a routine for checking the migratability of a thread, so one never attempts to migrate a non-migratable thread.

Another important characteristic is how much a thread contributes to the overall load of its processor. The overall load of a processor is equal to the sum of the loads of the individual threads residing on said processor. By default, all threads contribute equally to the overall load of the processor, and thus the default load of a newly created thread is 1. However, some threads are known to have more work to perform than others. Thus, Chant supports user specification of a thread's load at the time of its creation.

Often, it is impossible to predict how much work a thread has remaining, and thus the thread's load should never be altered. However, for some applications, a thread may have a specified amount of work to perform, such as a certain number of iterations. As the thread performs work, it knows how much work remains. Additionally, some applications may execute in a manner such that thread's go through cycles, where the amount of work remaining is not important, but the current amount of work available to perform varies with time. For situations like this, we provide a routine that allows a thread's load to be dynamically altered. For a thread that has a specific amount of work to perform, the user can regularly reduce the load associated with the thread. For a thread that has a varying amount of work to perform, the user can alter the load in a manner consistent with the amount of work the thread has available to perform.

An additional use of this functionality deals with blocked threads. While many suggest the length of the ready queue is a good indicator of load [37], this may not always be true. This would suggest that blocked threads should not contribute to the load of a processor.

However, if a particular application behaves in a manner such that threads do not typically block for a long time, it may be desirable to have blocked threads contribute some of their load to the processor's overall load. Since different applications will have threads that block for different durations, the amount a blocked thread contributes to the overall processor load should be controlled by the user. We have given the user this ability by allowing a thread's load to be dynamically altered. A user can lower a thread's load just prior to blocking, and raise it back to what it was after the thread is released.

We also provide a routine for obtaining the amount of load a thread contributes to the overall processor load.

A final thread characteristic that may affect the decision making phase is communication patterns. There are two main sources for performance degradation in a distributed memory parallel execution. The first, as we have been discussing, is load imbalance. The other important consideration is communication overhead. The more communication required for a parallel execution, the slower the execution is going to run. For this reason, it is often important to attempt to minimize communication overhead, while at the same time maintaining a balanced load. Unfortunately, these two issues can be orthogonal in many situations, in that balancing the load typically leads to increased communication, and vice versa. Therefore, it is important that a load balancing system attempt to minimize increased communication at the same time that it attempts to maintain a good system load balance.

In a multi-threaded environment, this translates to avoiding separating threads that frequently communicate with each other. Moreover, if load redistribution is necessary, it is preferable to move threads to processors with which they communicate often. That is, if thread A is doing a lot of communication with threads residing on the same processor as it resides, it is a poor choice for migration. If, however, thread A is doing a lot of communication with threads on processor X, it may be a good candidate for migration to X.

In order to support such a functionality, Chant keeps track of the communication histories of each thread within the system. Since threads are mobile, it would make little sense to keep this information separate from the threads themselves. Therefore, each thread has, as part of its TCB, a history of the number of messages that it has sent to each processor in the system. Additionally, Chant keeps track of the number of messages that a thread has received from each processor in the system. The load balancing API provides a routine for returning this information to the user. This information can then be used in the decision making phase in an attempt to minimize additional communication overhead.

While each of these characteristics can be obtained for a specific thread, it is often more useful to gather statistics relating to every thread on the run queue. Therefore, the routines for obtaining thread characteristics come in two forms. The first form is as discussed above, where a thread is specified and a single value (or set of values as in the case of communication history) is returned. The related routines, however, return the values of every thread on the run queue. These values are returned in an array, where the array index indicates the offset from the front of the run queue. That is, if the values are returned in array X, $X[0]$ is the value associated with the thread on the front of the run queue, while $X[i]$ is the value associated with the thread offset i places from the front of the run queue.

1.3 Global State Information

The previous section described routines that access and alter characteristics of local threads. However, many load balancing algorithms require global state information to make decisions. While it would be possible to make individual thread characteristics available to remote processors, this would introduce a tremendous amount of overhead. Therefore, we have made a conscious decision to allow remote processors to determine how much work should be moved where, but to require the processors from which the threads are migrating to determine which threads should be involved. That is, while processor A may inform processor B that processor B must move x amount of work to processor C, it is the responsibility of processor B to determine which threads should be migrated to processor C. Processor B would use the routines discussed above to make such determinations.

An important piece of information for many load balancing algorithms is the load of remote processors. For this, Chant provides routines that gather remote load information. One such routine is used to obtain the processor load of a specific processor. When this routine is called, the target processor is specified and a RSR message is sent to that processor. The calling routine then waits for a response containing the current processor load from the target processor. Another routine retrieves the loads of a set of processors. This set of processors can include all the processors in the computation, or just the processors defined in the calling processors load balancing domain, as discussed earlier. The initiating processor can use this information, along with its own processor load, to determine if a load redistribution is necessary, and possibly what the new load distribution should be.

Other information needed by some load balancing algorithms, is the communication history of certain processors. Though this information cannot aid in determining which threads need to be migrated from one processor to another, it can be used to help determine when a processor is doing a large percentage of its communication with threads on remote processors. This may indicate that it would be helpful to migrate work from this processor to the processors with which it is communicating. Since this information is not used by the default load balancing algorithm, it is included so it can be used by a customized load balancing algorithm. It is assumed the user has customized the other parts of the load balancing layer such that the choice of threads to migrate is made appropriately.

1.4 Manipulating the System

To this point, none of the routines we have discussed have any impact on the actual execution of an application. They have simply dealt with setting parameters and setting and gathering state information. Naturally, there are routines that affect the execution of an application. These routines deal with manipulating the system run queue, synchronization of processors, and the actual migration of threads.

Since a customized load balancing algorithm may access threads in any number of ways, we must provide mechanisms for manipulating the system run queue. We provide the user with three separate routines for removing threads from the run queue. The first routine allows the user to remove the thread at the front of the run queue. The other two routines allow the user to remove specific threads from the run queue. The first of these requires the user to specify an offset into the run queue to identify the thread. Thus thread *0* designates the thread at the front of the run queue while thread i would reside i spaces behind thread 0. The second way to specify the thread to be removed is to provide a pointer to the Chant TCB of the targeted thread. In addition to removing threads, threads can also be put back on the run queue. However, threads can only be added to the end of the queue.

It is important to note that these queue manipulation routines may affect the run queue extensively. For instance, there is no guarantee that the thread at the front of the run queue prior to executing one of these routines, will still be at the front of the run queue following the routine. This queue manipulation can clearly have major impact on the way an execution proceeds. Great care should be exercised in any use of these queue manipulation routines.

Certain load balancing algorithms may perform best if there is no computation being performed by any processors during the decision making phase. This can be especially true for complex algorithms. since other processors continuing computation may change the overall state of the system before the decision making phase is complete. In order to accommodate such algorithms, Chant supports the synchronization of all processors within the calling processor's load balancing domain. This routine instructs all the processors within the domain to stop computation and participate in a load balancing phase. This accomplishes two possible goals. One, it assures that there will be no change in the state of the system during the load balancing phase. Second, it can also be useful when the load balancing domain is small and it is beneficial to have quick responses to remote queries. For instance, when a query is sent to get load information from one of the synchronized processors, it is unnecessary to wait for the RSR server to return to the processor to respond to the query, since it is sitting on the processor, waiting to respond to queries. If the processors were not synchronized, the request would be queued on the remote processor, and not handled until the remote RSR server regains control of the processor. Even though there is still a latency associated with the message transfer and the processing of the query, with synchronization at least the query is handled immediately by the remote processor.

This synchronization has been implemented so as to ensure that two or more processors attempting to synchronize at the same time will never deadlock. This is accomplished by using two types of RSR messages, regular RSR messages and RSR messages associated with load balancing. When a processor is synchronized, it only receives RSR messages associated with load balancing. Likewise, any processor that has initiated a synchronization, receives query responses via the same format, receiving only RSR messages associated with load balancing. This allows a processor that has initiated a synchronization to acknowledge synchronization requests from other processors, and to respond to load balancing queries concurrent with its own load balancing phase. Furthermore, each processor keeps a tally of the number of outstanding synchronization requests it has handled. Thus, when a synchronization is released, the server only relinquishes control of the processor if there are no outstanding synchronization requests.

2 Load Balancing Routines

This section describes the middle sub-layer, which consists of various load balancing routines. These load balancing routines can be divided into two types: those that can be customized by the user, and those that are statically implemented. The statically implemented routines are those that actually carry out the load redistribution. The customizable routines, on the other hand, are those that constitute the decision making phase of load balancing. These decision making routines do not carry out any actual load redistribution, but rather, simply return values indicating the load redistribution that should be performed.

The first of these decision making routines would be called by a centralized load balancing algorithm. This routine takes as input the processor loads of each processor involved in the load balancing, be it the entire set of processors in the computation, or just those in the calling processor's load balancing domain. This routine returns a two dimensional array, say X, indicating how much load each processor should migrate where. The element *X[i,j]* represents the amount of work that processor i should migrate to processor j.

The default implementation of this routine simply attempts to balance the current load evenly across all processors. It does this by taking the average load of all the processors, and indicating that processors with load above this average should migrate an appropriate amount of work to those whose loads are below the average.

Let us use an example to explain the default implementation. Suppose we have an application running on four processors, *P0* to *P3*. Let the array $L = [5, 12, 8, 15]$ represent the current processor loads of the four processors. In this example the total system load is 40, which computes to an average of a load of 10 for each processor. The algorithm proceeds as follows:

- Processor *P0* has a current load of 5, which is below 10, so it does nothing.
- Processor *P1* has a load of 12, so it should move 2 units of work. Since processor *P0* has load 5, processor *P1* should move 2 units of work to processor *P0*. This will cause processor *P0* to have a load of 7.
- Processor *P2* has a load of 8, which is below 10, so it does nothing.

- Processor *P3* has a load of 15, so it should move 5 units of work. Processor *P0* has a load of 5, but is scheduled to receive 2 units of work form processor *P1*. Therefore, *P3* should send 3 units of work to *P0*, which would lower *P3*'s load to 12. Processor *P1* has a load of 12 and is scheduled to move 2 units of work to *P0*, so no work should be sent to *P1*. Finally, *P2* has a load of 8, so *P3* should send 2 units of work to *P2*.

If the redistribution is performed as indicated by the algorithm, each processor will have a load of 10 following redistribution.

Another of these decision making routines can be called by an under-loaded processor that needs to get work from other processors. This routine also takes as input the loads of all the processors involved in the load balancing. It returns an array that contains how much work should be requested from each processor. That is, if the return array is *X*, the value stored in *X[i]* is the amount of work that should be requested from processor *i*.

The default implementation simply chooses the most heavily loaded processor from the list of loads, and indicates work should be requested from it. The amount of work to be requested is the sum of the calling processor's loads and the most heavily load processor, divided by two. This would cause the two processors involved to have equal loads following the redistribution.

A related decision making routine can be called by an over-loaded processor that needs to send work to other processors. This routine also takes as input the loads of all the processors involved in the load balancing. It returns an array that contains how much work should be moved to each processor.

The default implementation simply chooses the least loaded processor from the list of loads, and indicates work should be sent to it. The amount of work to be sent is the sum of the calling processor's loads and the least load processor, divided by two. This would cause the two processors involved to have equal loads following the redistribution.

The decision making routines described above are used strictly for determining what load redistribution should be carried out. These routines have nothing to do with choosing which threads should be involved in the suggested redistribution. Choosing threads to be involved in a redistribution is a local decision. That is, the processors owning the threads are responsible for deciding which threads to migrate where. We provide additional decision making routines for this purpose. These are local routines that use thread characteristics, as described in the previous section, to select which threads should be involved in the load redistribution.

One of these local decision making routines determines which threads to migrate to a specific processor, based on a specified amount of work to be moved. It takes as input the amount of work that should be moved, and the destination processor. The output is a number indicating the number of threads to be migrated, along with an array indicating which threads should be involved. The output array designates threads based on their offset from the front of the run queue. This output array would be used as input to the redistribution routines discussed below. The default implementation simply traverses the run queue, selecting all migratable threads until the requested load has been reached.

The final decision making routine is related to the one just discussed. This routine decides which threads to migrate to various other processors, based on the amounts of work needed to be sent to the processors. This routine takes as input an array, say Z, which indicates how much work should be migrated to each processor. That is, *Z[i]* is the amount of work that should be migrated to processor i. There are three output values: The number of threads to be migrated, an array, say X, indicating which threads should be migrated, and an array, say Y, indicating the destination processor of the threads listed in X. That is, the value of *X[i]* is the offset from the front of the run queue of the ith thread to be migrated, and *Y[i]* is the destination processor of the thread indicated by *X[i]*.

This routine is called by the processor when a specific load redistribution is requested. The default implementation simply traverses the run queue, selecting all the migratable threads until enough work has been selected to satisfy the request for each destination processor.

The decision making routines discussed above simply decide which threads to migrate. These routines do not carry out the actual redistribution. Instead, the output from these routines is used as the input for various redistribution routines, which initiate the actual redistribution.

The redistribution routines can be separated into two types: remote redistribution routines, and local redistribution routines. The remote redistribution routines instruct remote processors to redistribute their load, while the local redistribution routines carry out the actual redistribution. The remote redistribution routines make use of Remote Service Requests, to instruct the remote processor to carry out the redistribution. The remote processors, upon receiving the RSR message, use one of the decision making routines discussed above to select which threads will migrate where in order to satisfy the request. These remote processors then make use of one of the local redistribution routines to perform the actual redistribution.

There are three different remote redistribution routines. The first of these instructs a remote processor to send various amounts of work to various processors. This routine takes as input an integer, say p, identifying the processor to initiate the redistribution, and an array, say X, which contains the amount of work to be moved to each destination processor. That is, the value of $X[i]$ is the amount of work that processor p should migrate to processor i. These input values are obtained from the output of the decision making routines discussed earlier.

The remaining two remote redistribution routines instruct remote processors to send work to the calling processor. One of these instructs a set of processors to each send various amounts of work to the calling processor. This routine takes as input an array, say X, that lists the amount of work to be requested from each processor. That is, the value of *X[i]* is the amount of work that processor i should send to the calling processor. This routine then sends RSR messages to each participating processor, requesting the indicated amount of work. The other routine that fits this category instructs a specific remote processor to send a specified amount of work to the calling processor. This routine takes as input an integer identifying the remote processor, say p, and an amount of work to be sent, say w. This will instruct processor p to send w amount of work to the calling processor.

This leaves the local redistribution routines, which initiate and actually carry out the migration of threads. The first of these send threads to a specific processor. This routine takes as input an integer, say p, indicating the destination processor, and an array, say X, indicating which threads should be migrated to p. The threads listed in X are identified as offsets into the run queue. Thus, this routine instructs the system to send all the threads in X to processor p.

A related local redistribution routine sends threads to various different processors. This routine takes as input two arrays. The first array, say X, lists the threads that should be migrated, while the second array, say P, indicates the destination processors for the threads listed in X. Thus, the thread at offset *X[i]* in the run queue should be migrated to processor *P[i]*.

Finally, we provide local redistribution routines, which perform the actual thread migration. The first of these routines allows a thread to migrate itself. This routine simply moves the thread to a target processor. The other local routine that fits this category is the one that actually migrates specific threads from one processor to another processor. This routine takes as input an integer, say n, indicating the number of threads to be migrated, along with an array of pointers to Chant threads, say T, indicating the threads to be migrated. Additionally, it takes as input an integer, say p, indicating the destination processor for the threads listed in T. That is, this routine migrates the n threads referenced by T to processor p. Upon return, the threads have been added to the run queue of the destination processor. When the migrated thread gains control of the destination processor, it proceeds with the same instruction it would have executed as if no migration had occurred. Typically, the threads referenced by T are not on the run queue at the time this routine is called. The caller must, in that case, return the threads to the run queue on the source processor, so that the underlying thread system can do some cleaning up following the migration. The system will know that the thread has migrated, so no additional effort is required of the user.

3 The Load Balancing Function

As mentioned above, the load balancing function is called by the system anytime it attempts to balance the system load. We provide a very simple default implementation for this function. If the local load (that is, the load of the processor calling the load balancing function) is below a user defined lower threshold, the processor loads of all the processors in the execution are gathered using the load gathering routine discussed in Chapter 6, Section 1.3. This array of loads is then passed to the decision making routine described in Chapter 6, Section 2, with a load redistribution being returned. This load redistribution is then passed to the remote redistribution routine described in Chapter 6, Section 2, which requests work from various processors, in accordance to the indicated redistribution. If none of the remote processors has enough work to share, load balancing is implicitly turned off. This is to prevent undue overhead associated with load balancing once the execution is close to terminating. That is, there would be fewer threads remaining than there are processors. In this case, no load redistribution will improve execution, and it will degrade performance if busy processors are forced to exchange load gathering messages with idle processors.

If, on the other hand, the local load is above a user defined threshold, the remote loads are gathered and passed to the global decision making routine intended for overloaded processors (as described in Chapter 6, Section 2). This routine also returns a load redistribution, which is used as input to the local redistribution routine that sends work to various remote processors (as described in Chapter 6, Section 2).

If the user has not registered customized decision making routines, this will be carried out using the default implementations. If, however, the user has customized the decision making routines, the default load balancing function will call the user supplied routines, which must return redistributions consistent in form with the default implementations. These results will be passed to the non-customizable redistribution routine that will perform the actual redistribution.

4 Customizability

As was pointed out at the beginning of this chapter, our goal is to provide a load balancing layer that can work as a framework within which users can make use of thread migration, and the provided API, to customize load balancing to fit their particular needs. We wish to allow the user to have near full control over the load balancing algorithm, allowing the amount of user customization to range from almost zero to near 100%. We have provided this capability through the load balancing API as well as supplying default implementations for all decision making routines. The customizability really arises from the ability of the user to replace these default implementations with their own customized implementations.

Users may decide that the load balancing algorithm provided through the default implementations serves their needs. In this case, no customization is performed, and the default load balancing function is called by the system, and this load balancing function makes use of the default implementations of the decision making routines. The only input required of the user in this scenario is the upper and lower thresholds, to be used by the default load balancing function, as well as how frequently the load balancing function should be called.

Conversely, the user may decide that the default implementations do not provide the desired functionality. This could be the case if a very complex algorithm is appropriate for the application. In this case, users can implement their own load balancing function and register it with the system. This causes the system to call this customized implementation each time it attempts to perform load balancing. Furthermore, this customized load balancing function can call user-level decision making routines, or the user can customize the default decision making routines and register those with the system as well. The user must still specify how frequently the system needs to call this customized load balancing function.

It is important to point out that these two scenarios, no customization and full customization, are not the only two choices. These provide the extremes of the range of customizability. The user has the option of mixing and matching choices from the two scenarios. The user may feel that the default load balancing function serves the application well, but that some or all of the decision making routines do not. In this case the user does not register a customized load balancing function, but does register customized decision making routines. Furthermore, some of the decision making routines may be customized while others are left with their default implementations. In such a case, the default load balancing function makes calls to the decision making routines, and the system ensures that the customized routines are called if they have been registered. It is the responsibility of the user to ensure that the customized routines return redistributions in a form consistent with the default routines. This is due to the fact that the default load balancing function will use the return values as input to the redistribution routines.

A customized load balancing function makes use of the lower level load balancing routines discussed earlier in this chapter, as well as the decision making and redistribution routines. Some of these decision making routines can also be customized, in which case they also would make use of the lower level load balancing routines. The user has control over customizing none, any, or all of the decision making routines as well as the option of customizing the upper most load balancing function. The ability to mix and match which routines are customized, gives the user extensive control over the load balancing algorithm, allowing it to be tailored to specific application.

This chapter has provided a detailed description of the load balancing layer built on top of Chant. We have concentrated on the decision making policies, with emphasis on the ability of the user to decide how much customization is desired. We have also introduced the default decision making functions and the API provided for customization. Chapter 7 provides results of performance testing. This will include quantifying the overhead associated with multi-threading and other functionalities, as well as the results of using the load balancing layer on a small suite of test applications.

Chapter 7

Performance

In this chapter we study the effectiveness of many of our decisions. We present the results of timing tests for the overhead associated with multi-threading, thread migration, and message forwarding. We then test our load balancing system on a variety of test applications under a variety of conditions. We present both the performance results, plus brief analyses of the overheads associated with the use of thread migration for the purpose of dynamic load balancing.

Unless otherwise stated, all tests were run on a dedicated network of four Sun SPARC-station 20 workstations, model 612 (dual 60 MHz SuperSPARC processors). Each machine was running Solaris 2.6 with full patches. Machine *P0* was the OS server for the rest of the machines, all of which were auto clients. The experiments were set up with only a single user process running on each processor. so that the tests would not be affected by any other processes except normal system services. The machines were interconnected via a private FDDI Network. and thus there was no external network traffic. We used the MPICH version 1.0.13 implementation of MPI with debug flags turned off during compilation.

All the performance numbers were collected by taking the average of ten executions.

1 Overheads Associated with Multi-threading

There is certain overhead associated with any multi-threaded application. Here we attempt to quantify the overhead associated with various operations in multi-threaded applications.

In a multi-threaded application. the most common overhead is that associated with switching control of the processor from one thread to another. This is referred to as context switch time. Additionally. our system requires the registering and releasing of pointers. for the purpose of supporting thread migration. This introduces additional overhead to a multi-threaded application. Finally. Chant supplies its own implementation of dynamic memory allocation and free routines. For any application that does frequent dynamic memory allocation. we must analyze the difference in time required to perform these different routines. The memory allocation times reported here are for the block size required for our implementation of the Traveling Salesman Problem. which is 148 bytes.

Routine	Time (microseconds)
Creation-termination	1297.77
Context switch	32.36
Register/release	2.14
System Malloc/free	3.03
Chant Malloc/free	2.09

Table 7.1: *Overheads associated with various thread operations*

Table 7.1 shows the overhead associated with these various routines. The numbers for memory allocation and deallocation are an artificial benchmark. gathered by simply looping through a number of iterations doing malloc/free pairs. Therefore, these times are only approximate as these numbers can be influenced greatly by a number of different factors. including memory allocation history. Also, the Chant malloc/free pair performs significantly better than the system malloc/free pair. This is because Chant it doing simple memory management and has a lot less memory information to maintain than does the system.

As can be seen from this table, the most significant overhead can be attributed to context switch times. For this reason, it is important in any multi-threaded application to limit the total number of context switches. These numbers will be used in the analysis of performance in the following sections.

2 Application Termination Detection

In a typical multi-threaded application running on multiple processors, the application program knows it has finished its computation for a particular processor when all the threads terminate on said processor. However, when threads can migrate between processors, this is not sufficient. That is, just because all the threads have terminated, the processor may be required to do additional computation if threads migrate to the idle processor. For this reason, any application program must implement some type of termination detection on its own. This is on top of the termination detection that has already been implemented within Chant.

Each of the test application discussed later in this chapter use an additional user-level thread per processor for this purpose. This thread is created at startup, and its sole purpose is to determine when the entire computation is completed. The thread checks to see if there are any other user-level threads on its processor. If there are, it blocks on a mutex. If there are no other user-level threads, it goes into a busy wait, trying to receive a message from its neighboring processor. This message is referred to as a token. While this thread waits for the token to arrive, it continually checks to see if there are any other user-level threads on the processor. If a thread migrates to this processor, this termination detection thread will find it, and block again on its mutex.

When a user-level thread terminates, it checks to see if it is the last user-level thread on the processor. If it is not, it simply terminates normally. However, if it determines it is the final user-level thread, in unblocks the termination detection thread. When the termination detection thread wakes up, it sets its state to *active*. Upon receiving the token, its sets the token value to its state, sets its state to *idle*, and sends the token to the next processor. If the termination detection thread's state is *idle* and it receives the token as being *idle*, it knows all the processors have terminated. This is a variation of Dijkstra's algorithm for termination detection [47].

The use of this termination detection thread adds a small amount of overhead to the computation, but is essential for proper termination when thread migration is possible.

3 Thread Migration Performance.

We have implemented thread migration such that the system can migrate a thread to another processor at arbitrary points during the thread's execution. It is our goal to show that threads can be migrated across processor boundaries in an efficient manner with low overhead. We have tested the time it takes to migrate threads across processor boundaries as well as the time spent forwarding messages. This section presents these test results along with an analysis of the performance.

3.1 Migration Times

In order to analyze the time it takes to migrate a thread from one processor to another, two sets of tests are conducted. The first set of tests uses MPI primitives with no multi-threading. In this set of tests, machine *P0* initiates a message of size m, and sends it to machine *P1*. Each processor in turn receives the message and forwards it to the next processor. The last processor forwards it back to *P0*. Each test uses four processors and consists of sending the message around this cycle 100 times. The total time to complete the test is divided by 400 to give the average message passing time. The size of the message, m, varies with each test run, giving us the average message passing time for various message sizes.

The second set of tests creates a thread with stack size m on machine *P0*. This thread migrates to machine *P1*, which receives it and puts it on a queue. *P1* then removes the thread from the queue and migrates it to machine *P2* which enqueues it, dequeues it, and migrates it to machine *P3*. *P3* does the same and migrates the thread back to *P0*. This loop is carried out 100 times on four processors, and the total time is measured. This total time is again divided by 400, to get the average migration time for a thread with stack size m. Stack size m varies using the same values used in the first set of tests.

Table 7.2 shows the results of these tests where *Size* is the size of the message or the thread's stack. *Communication* is the average time to send a message of size m using

MPI primitives (first set of tests), while *Migration* is the average time it takes to migrate a thread with a stack of size m (second set of tests). *Overhead* is simply the difference between *Communication* and *Migration*. The times are all in milliseconds.

Size (m)	Communication	Migration	Overhead
16K	4.1	7.6	3.5
32K	8.2	12.2	4.0
64K	14.3	18.4	4.1
128K	25.7	30.5	4.8
256K	49.8	54.1	4.3

Table 7.2: *Thread migration time (in ms) with varying sized stacks*

Since the migration takes place in several steps, we tried to account for this overhead. Migration is accomplished with four messages being sent to the destination processor. These messages are: an integer indicating the size of the thread's stack, the underlying thread packages thread control block (TCB), Chant's TCB, and the thread's stack. The underlying TCB has a size of 72 bytes while Chant's TCB has a size of 4400 bytes. The size of the Chant TCB is dependent upon the maximum number of processors used in the application and the maximum number of message tags that can be used for message passing, whose values were 8 and 100 respectively, for these experiments. These first three messages together take approximately 3.0 ms to send.

These numbers show that other than communication overhead, which cannot be avoided, there is very little overhead associated with the thread migration. The small additional overhead (ranging from 0.5 ms to 1.8 ms) can be attributed to table traversal for updating pointers and a list traversal for reposting pending receive operations. This is a small price to pay for supporting pointers and communication for migratable threads.

3.2 Message Forwarding

The previous section only studied thread transmission overhead and overhead incurred by the destination processor. This section presents the overhead associated with forwarding messages, i.e., the time it takes the source processor to forward messages to a migrant thread. These tests were run on 2, 4, and 8 processors, with a varying number of messages needing to be forwarded. Each message that was forwarded was one kilobyte in size.

	Number of processors		
Messages	2	4	8
No messages	5.5	8.1	13.7
1 message per proc	6.6	10.9	21.5
2 message per proc	7.8	13.6	28.6
3 message per proc	9.6	18.4	41.8
per message	+0.51	+0.64	+0.88

Table 7.3: *Forwarding overhead and per message time (in ms.)*

Table 7.3 shows the results of these tests, where the first row is the amount of time spent when there are no messages that need to be forwarded. This is overhead associated with the forwarding algorithm itself. These numbers increase with the number of processors since each processor must be handled separately. The next three rows show the total time needed to forward all messages with 1, 2, and 3 messages per processor, respectively. The total number of messages forwarded is equal to the number of messages per processor multiplied by the number of processors. The final row indicates the average time per message for forwarding. The average amount of time it takes to forward a message increases with the number of processors. This is due to the fact that the MPI implementation we used takes longer to retrieve messages from the system buffer when there are more processors.

These times only reflect the extra time spent by the source processors. They do not take into account any effects of increased network contention or increased work on other processors. Still, these numbers show acceptable overhead associated with the forwarding algorithm, and low overhead associated with the actual forwarding of the messages.

4 Test Applications

This section presents the overall execution times for our set of test applications. Each application has four versions:

- A sequential, non-threaded version
- An MPI based, parallel, non-threaded version
- A parallel, multi-threaded version, with no load balancing
- A parallel, multi-threaded version, with load balancing

4.1 Traveling Salesman Problem (TSP)

The first application is a traveling salesman problem. Since we are only interested in how well our load balancing layer performs, we used a very naive algorithm, unconcerned with the relative performance of the algorithm. We tested the algorithm on three sets of data, or graphs, each consisting of 17 cities. The first graph produces a perfectly balanced search tree, allowing us to study the overhead associated both with multi-threading, and with load balancing. The second graph produces a search tree that only searches a single branch of the tree. This causes the computation to be as unbalanced as possible, forcing all the computation to a single processor. This allows us to demonstrate the speedup achievable by thread migration in a best case scenario. The final graph was randomly generated. This allows us to study the effectiveness of thread migration for load balancing in what might be an average case TSP.

The multi-threaded version is set up such that the first 15 computation threads consider a path starting at city *0*, with city *1* being the first city in the path. Each of these threads designates a different city as the second city in the path. The next 15 computation threads consider a path starting at city *0*, with city *2* being the first city in the path. Each of these threads designates a different city as the second city in the path. This pattern continues for each of the 16 cities as the first city visited, for a total of 240 threads. These threads are spread evenly across the available processors.

Each of the computation threads, if allowed to, could run to completion. However, since we are attempting to perform dynamic load balancing, we must assure that there are opportunities to redistribute the load. This can only be done by the *server thread.* Thus we must perform a reasonable number of context switches. This requires making a decision regarding how often a context switch should be performed. The more often we switch threads, the more often we can attempt load balancing, and thus we can attain finer-grained load balancing. This may suggest that we switch threads following every iteration. However, each of these test cases performs tens of millions of iterations, and as noted earlier, context switches are expensive operations. Therefore, it would be unreasonable to switch threads following each iteration. We must find a comfortable compromise. In these tests, we decided to switch threads every 10,000 iterations. This requires keeping a count of the number of iterations, and performing a comparison following each iteration, to determine if a context switch should be performed. This extra comparison adds additional overhead to the multi-threaded application. In fact, each comparison takes approximately 0.55 microseconds to perform. This extra time must be taken into account when analyzing the performance of the multi-threaded code.

The non-threaded parallel version is implemented such that each processor periodically checks to see if another processor has found a solution better than the previous best. This means, when a processor finds a solution better than a previous best, it broadcasts this solution to the other processors. We use *MPI_Test* to check to see if a message has arrived. This can be an expensive operation, so once again we do not want to make this call following every iteration. We decided to check for a message every 10,000 iterations, introducing the same overhead to the parallel non-threaded implementation as we discussed for the multi-threaded version.

Tables 7.4, 7.5, and 7.6 show the results for a 17 city traveling salesman problem. The first column shows the application, while the remaining columns show the running time in seconds for a single processor, two, four, and eight processors. Figures 7.1, 7.2, and 7.3 show the same results in graphical format. The *x-axis* is the number of processors, while the left side *y-axis* is the execution time in seconds, and the right side *y-axis* is the parallel speedup, computed with the sequential version as the base.

Table 7.4 and Figure 7.1 show the results for a perfectly balanced search tree. This is a worst case scenario for load balancing due to the fact that there is no load imbalance, and therefore performance improvement is not available. This particular search tree performs about 25.4 million iterations for both the multi-threaded and non-threaded implementations, with about 2660 context switches in the multi-threaded implementation.

If we look at the sequential version vs. the MPI based non-threaded version on a single processor, we see that the MPI version runs slower than the sequential version. This is due to the additional overhead associated with the comparison operations discussed above. This overhead accounts for about 14 seconds with 25.4 million iterations.

	Number of processors			
Implementation	1	2	4	8
Sequential non-threaded	216.1	—	—	—
MPI based non-threaded	231.3	121.3	61.3	31.0
Multi-threaded with no load balancing	241.0	120.8	60.4	31.5
Multi-threaded with load balancing	241.7	120.9	61.0	32.6

Table 7.4: *Execution time (in sec) for perfectly balanced 17 city TSP*

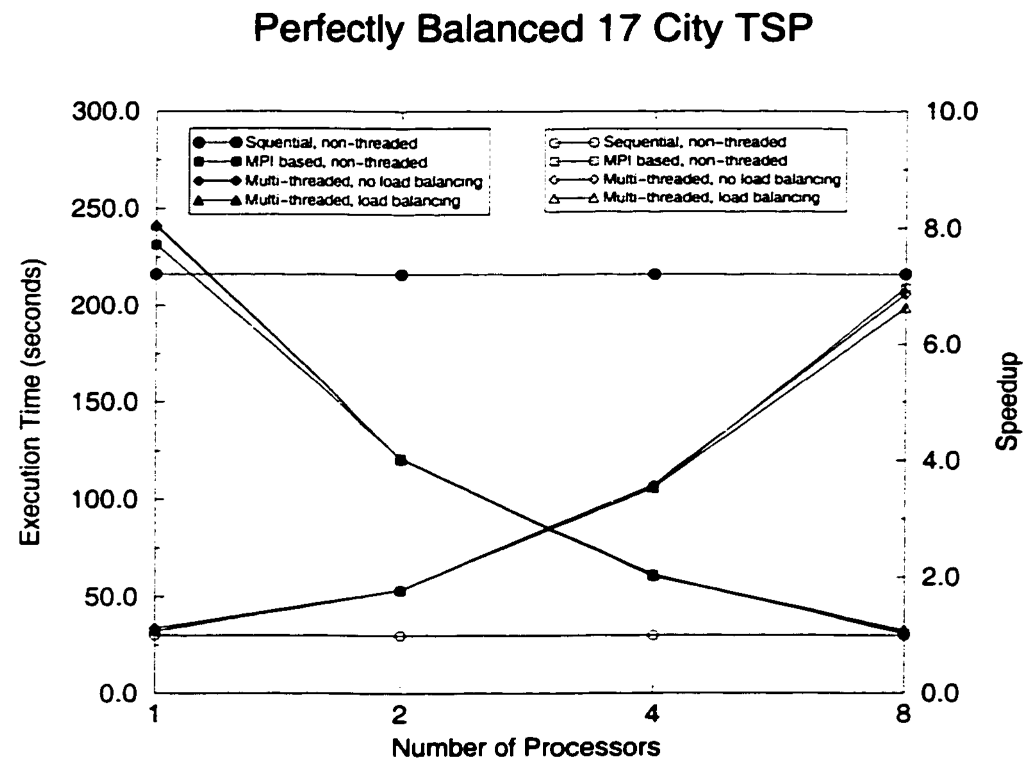

Figure 7.1: *Execution time and speedup for perfectly balanced 17 city TSP*

When we factor this into account, as well as the fact the even identical processors don't compute at precisely the same speed from execution to execution, we still see the MPI based version running slower than we would hope. After considerable work trying to quantify the additional execution time, we have concluded that it is due to cache effects. We suspect that parallelizing the code alters the memory access patterns in such a way as to introduce this additional overhead. This is supported by the fact that when we look at the times for 4 and 8 processors, compared to the time on 2 processors, we see near optimal speedup. It is impossible to quantify these effects without the help of profiling tools, which were unavailable to us.

When we compare the sequential time to the multi-threaded times, we must take into account that the multi-threaded code is paying the cost of the aforementioned comparison on a single processor, while the sequential code does not. This accounts for about 14 seconds of overhead. Additionally, the multi-threaded code must register and release pointers. Each iteration accounts for one register/release pair, though some iterations do multiple registrations while other iterations do no registration. According to our measurements, the registration/release costs contribute about 54 seconds to the overhead for these tests.

Finally, we must take into account the fact that memory allocation/deallocation is faster in Chant. However, using the benchmark presented in the previous chapter, we find that the multi-threaded version has less overhead than expected. This can be attributed to the fact that memory management can be affected greatly by various factors. Therefore, we conclude that memory management in the non-threaded version is costlier than our estimate.

When we study Figure 7.1, as should be expected, the multi-threaded version provides excellent speedup when we move to 2, 4, and 8 processors. Additionally, we see that very little overhead is introduced to the multi-threaded code when we enable load balancing. It is encouraging that enabling load balancing produces minimal overhead in an execution that offers no opportunities for load redistribution.

Finally, we see that the parallel multi-threaded versions run in about the same time as the parallel non-threaded versions. This is surprising since we know there is certain overhead associated with multi-threading. However, this can be attributed to the fact that memory management times vary greatly under different circumstances.

Table 7.5 and Figure 7.2 show the results for a radically unbalanced search tree. This is a best case scenario for load balancing due to the fact that the load is imbalanced to the point where all the computation is performed on a single processor. This particular search tree goes through about 34.6 million iterations for both the multi-threaded and non-threaded implementations, with about 3960 context switches in the multi-threaded implementation.

	Number of processors			
Implementation	1	2	4	8
Sequential non-threaded	296.9	—	—	—
MPI based non-threaded	317.2	317.1	317.3	317.2
Multi-threaded with no load balancing	327.6	326.2	324.2	326.8
Multi-threaded with load balancing	328.4	171.8	95.6	65.3

Table 7.5: *Execution time (in sec) for completely unbalanced 17 city TSP*

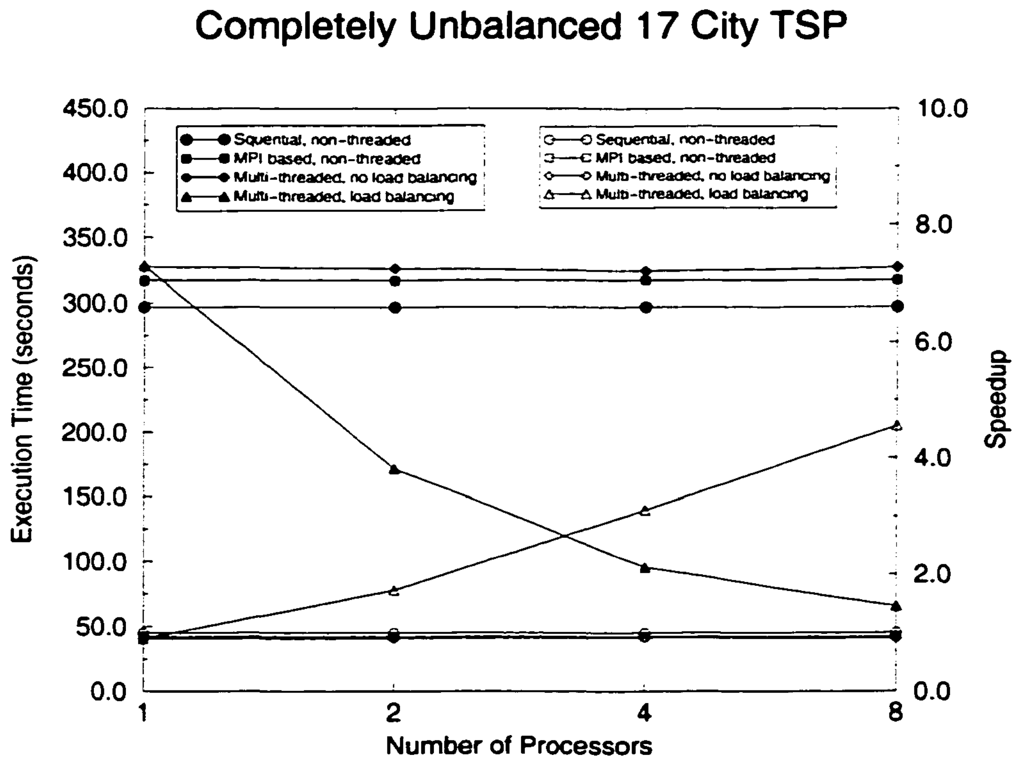

Figure 7.2: *Execution time and speedup for completely unbalanced 17 city TSP*

One of the first things we notice is that the non-threaded execution time jumps when we go from the sequential version to the MPI based version running on a single processor. This can be attributed to the comparisons discussed earlier. These comparisons contribute approximately 19 seconds to the execution time. When this overhead is added to the sequential time, we see that the MPI based version on a single processor runs in about the same time as the sequential version. We see no speedup when we go to multiple processors for the MPI based version. This is what we would expect since the load is imbalanced to the point that no speedup is available.

Once again we see significant, yet reasonable, overhead associated with multi-threading. Again, however, we see that the overhead is less than what would be expected when using the benchmarks discussed earlier. This can, of course, again be attributed to the fact that the memory management overhead varies greatly from the artificial benchmarks discussed previously.

When we do not enable load balancing in the multi-threaded code, we see no speedup. This is again due to the radical imbalance of the search tree, and is what would be expected.

However, when we enable load balancing, we see significant speedup. We do not see near optimal speedup for a couple of reasons. One reason is that there is inherent overhead involved with the thread migration, as discussed earlier. Another reason is that even though there are many threads that must perform a significant amount of computation, not all of these threads perform the same amount of computation. Since the amount of work a thread must perform is unknown at startup, and the amount of computation remaining is unknown at runtime. we cannot perform ideal load redistributions. Instead. we simply randomly select threads to migrate. Though this method does not achieve optimal speedup, it performs significantly better than not using any load balancing technique.

Table 7.6 and Figure 7.3 show the results for a randomly generated search tree. This represents an average case scenario for load balancing, since some branches will be pruned early while others will continue deep into the tree.

In the examples discussed above, each implementation performed the same number of iterations. This is because those test cases were constructed for the sole purpose of being either balanced or unbalanced. For this reason, all branches of the search tree were either searched to a leaf, or pruned at the root. Therefore, if a new solution was found, it was always equal to the best solution, and thus had no effect on any other parts of the computation. This means that it did not matter in what order the branches were searched.

Once we move to the randomly generated cases, however, this situation changes. Since we are using a naive algorithm, which does a simple depth first search, the order in which the branches are searched can affect the amount of computation. This is due to the fact that a branch that may be searched late in the computation on a single processor may be searched much earlier in a parallel run. If this branch leads to a good solution, it may cause

	Number of processors			
Implementation	1	2	4	8
Sequential non-threaded	1332.4	—	—	—
MPI based non-threaded	1418.9	814.6	477.2	344.4
Multi-threaded with no load balancing	1677.4	877.0	526.6	278.3
Multi-threaded with load balancing	1678.4	847.4	449.6	243.3

Table 7.6: *Execution time (in sec) for randomly generated 17 city TSP*

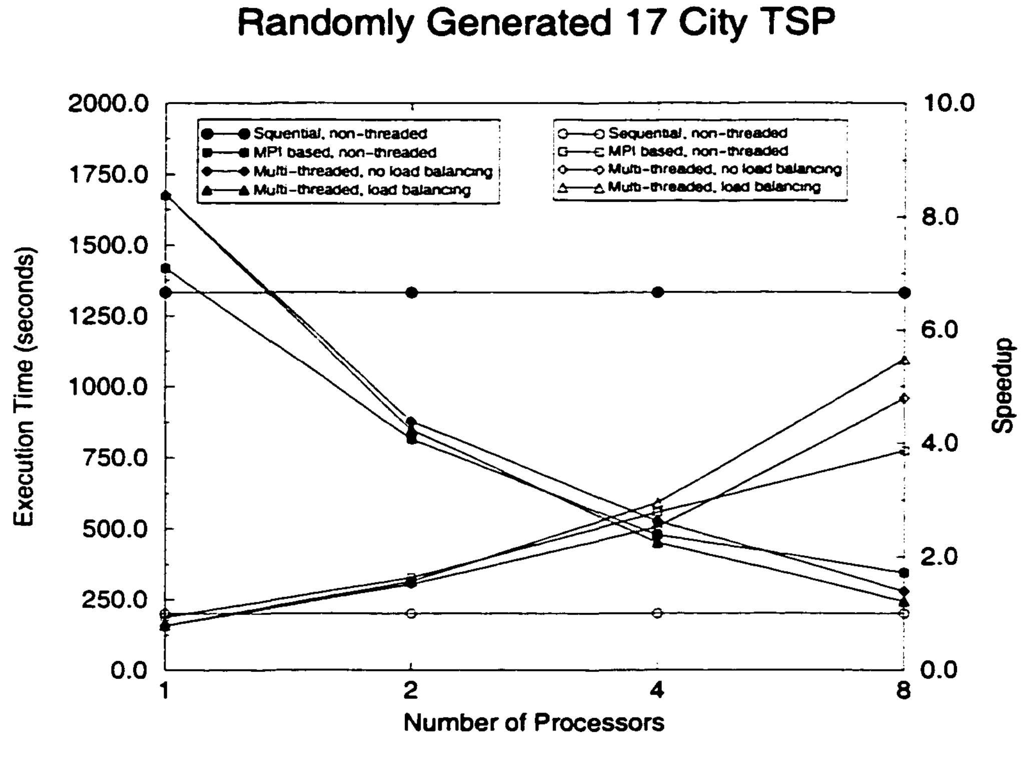

Figure 7.3: *Execution time and speedup for randomly generated 17 city TSP*

branches on other processors to be pruned earlier than they would have been otherwise. This would lead to fewer iterations. and thus less computation. It is important to point out however. that the opposite effect is equally as likely. That is. parallelizing the code is as likely to accelerate the finding of a good solution as it is to delay the finding of a good solution. in a relative sense.

Let us illustrate this concept by assuming we have two different search trees. With the first search tree. let's assume that the best solution can be found on the first branch of the second half of the tree. In a sequential search. this solution will not be found until after the first half of the tree has been searched. allowing pruning of the remainder of the tree. If we parallelize this for two processors. however. the second processor will find this solution immediately. and will send this result to processor 1. This will allow processor 1 to prune much of the first half of the tree. thus decreasing the amount of computation required.

Now for the second tree. let's assume the best solution can be found on the last branch of the first half of the tree. In a sequential search this solution will be found in about the same amount of time as for the first tree. again allowing pruning of the second half of the tree. If we parallelize this for two processors. the first processor will not find this solution until it has searched the entire first half of the search tree. while the second processor is busy searching the second half of the search tree. Since the best solution is not found until the end of the search. there is no opportunity to prune the second half of the tree accordingly.

This results in the work being done by the second processor being wasted, and only minimal speedup may be achieved.

This can also be seen in a multi-threaded approach on a single processor. Since there are many threads searching the same tree, better solutions may be found earlier than without multi-threading. Again, multi-threading is as likely to accelerate the finding of a good solution as it is to delay the finding of a good solution, in the same way as described above.

Theoretically, there should be no increase in iterations for the multi-threaded code with additional processors. This is because the threads are searching the branches in the same relative order as for the single processor case. In practice, however, there should be a slight increase in iterations as we add processors. This is because of the latency involved in communicating new best solutions to other processors. That is, when a thread in the multi-processor case finds a new solution, it must communicate this new value to the other processors. The time during which this communication is taking place is used by the other processors to continue searching the tree. This causes them to search branches that need not be searched, simply because the new best solution is not yet known.

	Number of processors			
Implementation	1	2	4	8
Non-threaded	144.9	160.2 (83.3)	162.0 (48.7)	164.8 (36.1)
Multi-threaded with no load balancing	173.4	172.5 (89.2)	177.6 (54.0)	177.0 (29.1)
Multi-threaded with load balancing	173.4	171.8 (87.7)	183.7 (47.1)	179.8 (25.2)

Table 7.7: *Number of iterations for randomly generated 17 city TSP (in millions)*

Table 7.7 shows the number of iterations required for the different executions. For the multi-processor executions, the second number is the number of iterations performed by the most heavily loaded processor. We can see from the numbers that multi-threading the code causes additional workload for both the single processor and multi-processor examples. This should be taken into account when analyzing the performance. We also see that the non-threaded code performs more total iterations when it is parallelized. This can be attributed to the effects discussed earlier.

Looking back at Table 7.6 we see that the multi-threaded code takes significantly longer than the non-threaded code to complete on a single processor. This can be attributed to overhead associated with multi-threading, but most of the additional time is due to the fact that the multi-threaded code performs approximately 28 million additional iterations. As we discussed, there is always a chance that we will see additional computation when we alter the order of the search.

Table 7.7 also shows us that enabling load balancing for the multi-threaded case does in fact produce a balancing out of the load. In all three cases (2, 4, and 8 processors) we see that enabling load balancing causes the most heavily loaded processor to perform fewer iterations. This is what would be expected since we are migrating threads from the most heavily loaded processor. However, the fact that we achieve lower execution times with load balancing enabled suggests that the overhead associated with thread migration is low enough that the performance gains outweigh the costs.

Additionally, we see that enabling load balancing gives us improved performance over the non-threaded version for both 4 and 8 processors. This shows that the performance gains we achieve by load redistribution outweigh the costs associated with multi-threading. In fact, we see a 29% improvement over the non-threaded case when we enable load balancing on 8 processors.

4.2 Adaptive Quadrature

The next application is an adaptive quadrature application using Simpson's algorithm. We tested the algorithm on two different integration functions. The first function is

$$f(x) = 1000 * sin(3000 * x) \tag{1}$$

where x is in radians. We integrated using an epsilon of $10 * 10^{-10}$ and an integration interval of [0,1]. This function is a simple oscillating function that converges at the same rate within all integration intervals. This should allow for a perfectly balanced parallel execution.

Figure 7.4 shows a graph of Function 1 for the interval [0,0.1].

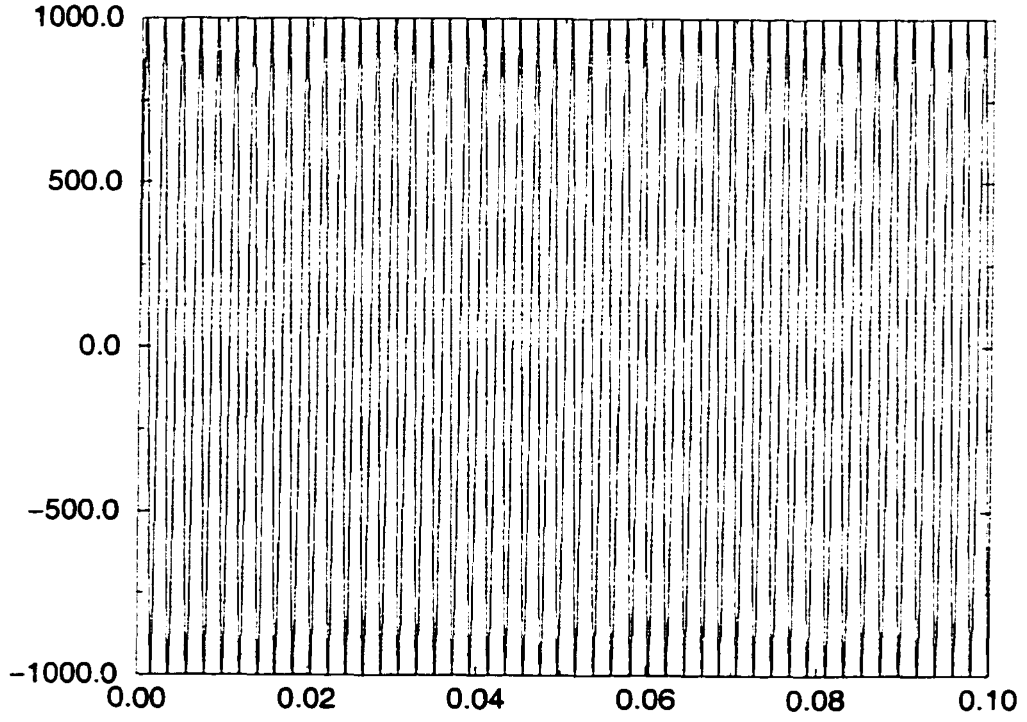

Figure 7.4: *Function 1 on the interval [0,0.1]*

The second function we integrated was

$$f(x) = sin(100 * x) + ((x/14)^{100} * sin(3000 * x^2) \quad (2)$$

where x is in radians. We integrated using an epsilon of $10 * 10^{-4}$ and an integration interval of [0,16]. This is an oscillating function that is very flat on the left part of the integration interval, but which varies greatly as the value of x increases. This causes the integration to converge quickly when we are close to 0, and much more slowly as we approach 16. This allows for a very unbalanced parallel execution.

Figure 7.5 shows a graph of Function 2 for the interval [0,16].

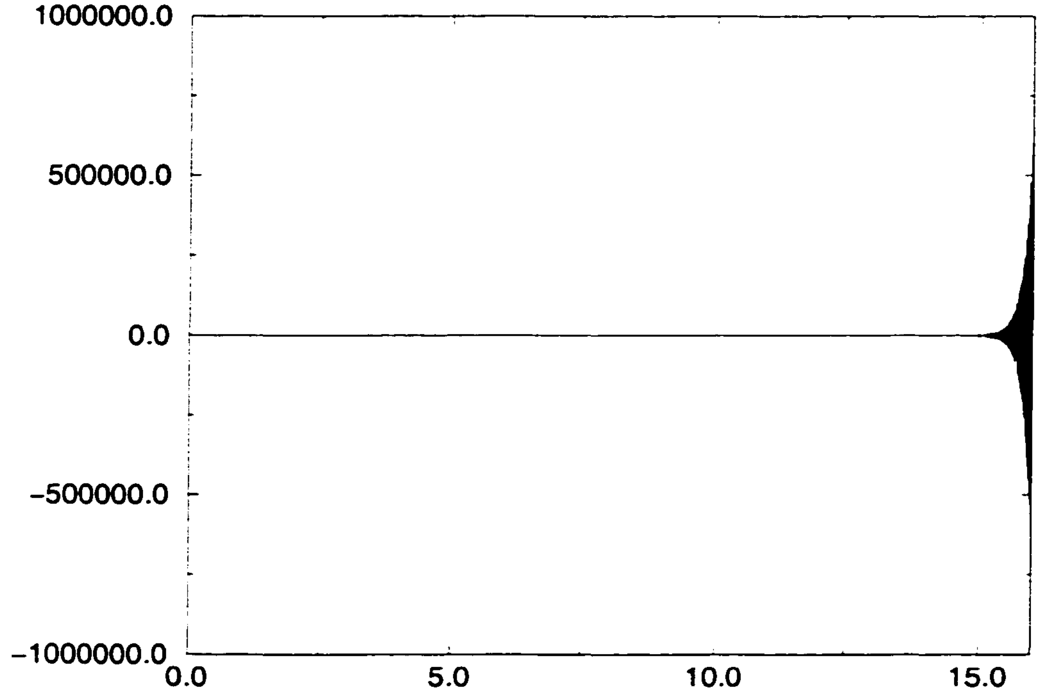

Figure 7.5: *Function 2 on the interval [0,16]*

The sequential version uses Simpson's algorithm in a straightforward manner. The MPI based parallel non-threaded version divides the integration interval evenly across the available processors. Each processor will run to completion, with no interaction from other processors. Therefore, there is no interaction between processors, and there is no direct overhead associated with parallelizing the code. The multi-threaded version divides the integration interval evenly across the available processors, and then divides the sub-intervals evenly across the threads created on each processor. Each processor creates 64 threads for the multi-threaded version.

As in the traveling salesman problem, each of the computation threads, if allowed to, could run to completion. Again, since we are attempting to perform dynamic load balancing, we must assure that there are opportunities to redistribute the load. We must, therefore, decide once again how often we should perform a context switch. For these tests we perform a context switch every 1000 iterations.

Table 7.8 and Figure 7.6 show the results for integrating Function 1 across an interval of [0, 1]. As mentioned above, this function is a simple oscillating function that converges at the same rate over all areas of the integration interval. This allows for a very balanced parallel implementation, and there is little or no performance gain available from load balancing.

We can see from the table and graph that parallelizing the code introduces no additional overhead, which is what was expected since there is no interaction between processors in the parallel implementation. Additionally, we see near optimal speedup from the parallel executions. Again, this is expected, since each processor has the same amount of work to perform, namely $1/p$ the amount of work as the sequential version, where p is the number of processors.

When we turn our attention to the multi-threaded implementation, with load balancing disabled, we see a small amount of slowdown. This can be attributed to a small degree to the overhead associated with multi-threading. The majority of this slowdown, however,

	Number of processors			
Implementation	1	2	4	8
Sequential non-threaded	163.4	—	—	—
MPI based non-threaded	163.4	82.4	41.3	20.8
Multi-threaded with no load balancing	172.3	86.8	43.5	22.8
Multi-threaded with load balancing	172.7	87.3	44.5	23.2

Table 7.8: *Execution time (in sec) for Simpson's algorithm integrating Function 1*

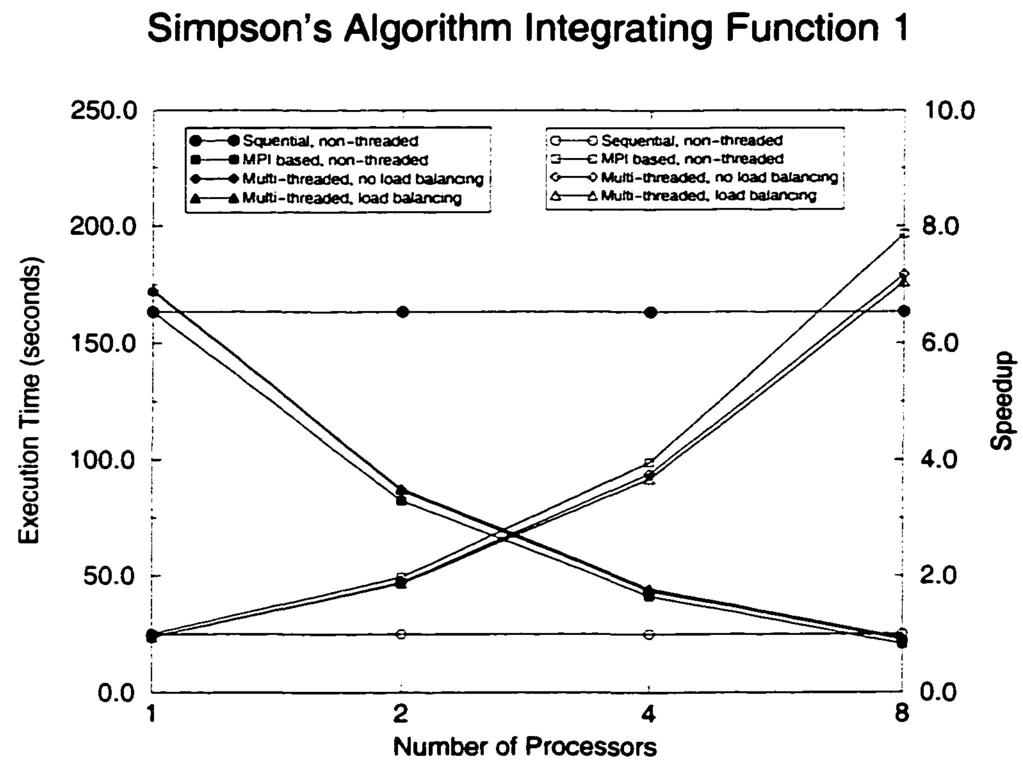

Figure 7.6: *Execution time and speedup for Simpson's Algorithm integrating Function 1*

can be attributed to the cost of the comparisons discussed above. After each iteration, the multi-threaded code checks a counter to see if it should perform a context switch. The time spent performing this comparison accounts for the majority of the overhead. We again see near optimal speedup (relative to the single processor multi-threaded execution time) when we parallelize the multi-threaded execution. Again, this is as expected.

When we study the times for the multi-threaded implementation with load balancing enabled, we see very little fluctuation from the times with load balancing disabled. The small difference in these times can be attributed to overhead associated with going through the load balancing phase when there is no redistribution needed. Since the computation is perfectly balanced, there is never an attempt to migrate threads, therefore execution times should be consistent regardless of if load balancing is enabled or disabled.

Table 7.9 and Figure 7.7 show the results for integrating Function 2 across an interval of [0, 16]. As mentioned above, this function is an oscillating function that is very flat in the lower end of the integration interval, and becomes much more pronounced as we move across said interval. This allows for a very unbalanced parallel implementation, with a great deal of speedup available from load balancing.

The results show us once again that parallelizing the sequential code introduces little or no overhead, as the sequential, non-threaded code, runs in about the same time as the MPI based non-threaded code running on a single processor. Additionally, we see

	Number of processors			
Implementation	1	2	4	8
Sequential non-threaded	216.4	—	—	—
MPI based non-threaded	216.5	216.5	216.5	213.5
Multi-threaded with no load balancing	222.9	223.2	223.4	221.6
Multi-threaded with load balancing	222.9	126.6	80.2	50.3

Table 7.9: *Execution time (in sec) for Simpson's algorithm integrating function 2*

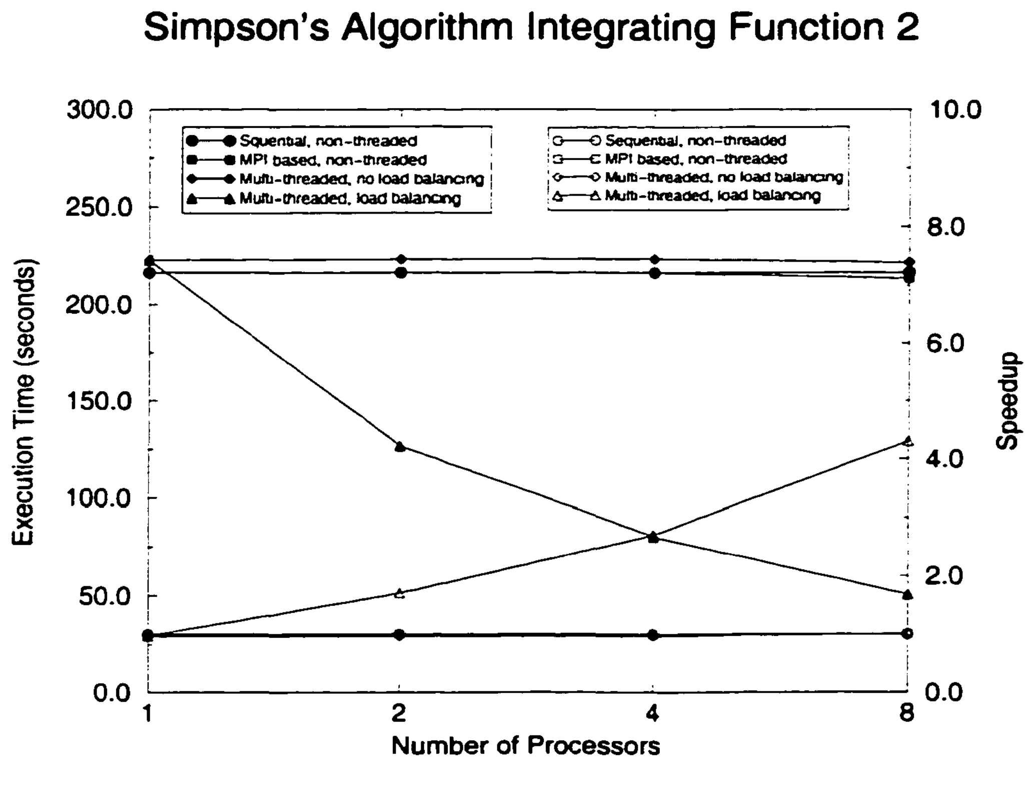

Figure 7.7: *Execution time and speedup for Simpson's Algorithm integrating Function 2*

that parallelizing the non-threaded code achieves no speedup for up to 4 processors, and negligible speedup on 8 processors. This is due to the fact that the integration is extremely unbalanced running in parallel.

When we study the execution times for the multi-threaded implementation with load balancing disabled, we again see a small amount of overhead associated with the comparison used to determine when a context switch should be performed. Other than this overhead, the multi-threaded version performs in much the same way as the non-threaded version, with no speedup for up to 4 processors and negligible speedup for 8 processors.

When we enable load balancing in the multi-threaded executions, however, we see significant speedup. This is due to the fact that most of the processors finish integrating across their sub-interval almost immediately, and then ask for additional work from processors that are still busy. The executions do not approach optimal speedup due to the overhead associated with migrating the threads. The overloaded processors lose some processing time to perform the migration, while the under-loaded processors are idle while they wait for new threads to arrive. Additionally, since we cannot predict when a thread will converge, all threads have the same load associated with them, with no estimation of how much work remains. This causes instances where a thread is migrated though it is about to converge. This is not advantageous since more time is spent migrating the thread than would have been required for the thread to terminate on its own. However, the costs associated with thread migration are far outweighed by the gains attained by load redistribution.

4.3 Volume Rendering

The final application is a volume renderer using a real life set of volume data. We ran two sets of tests, each using different opacity maps. The opacity maps determine how transparent or opaque the image is. The first set of tests were ran with an opacity map that made the image almost completely transparent. This allows all of the rays to traverse the entire volume, making for a very balanced parallel execution. The second set of tests used an average opacity map. which allows for some of the rays to terminate more quickly that others. This can lead to a load imbalance.

There are two issues that should be discussed before we look into the execution times. The first issue has to do with cache effects. The chosen algorithm uses a large amount of volume data for the rendering stage. This volume data does not fit into cache. The parallel execution breaks this volume up among the available processors, affecting the cache locality. This causes the code to run faster in parallel. This means that for a very balanced rendering, we can expect super linear speedup. Additionally, if a parallel execution shows excellent speedup, we must account for the cache effect. That is, a very unbalanced rendering will show speedup as if it were fairly balanced.

This also affects the multi-threaded implementation. Since each thread has a smaller set of volume data, threads have better cache locality. This means that a multi–threaded version may run faster on a single processor than the sequential version.

The second issue involves the way the executions are parallelized. We used an existing algorithm that simply divides the volume evenly among the processors. Each processor then renders a sub-image based upon this sub-volume. It does this by calculating the part of the image space that is involved with its assigned sub-volume, and tracing each pixel within this image space. This is done with no knowledge of the rest of the volume data. This can affect the rendering times by causing rays to be traced in the parallel version beyond the point where they terminate in the sequential version. This causes more actual computation to be performed in the parallel execution. We will attempt to explain this better using a couple of figures.

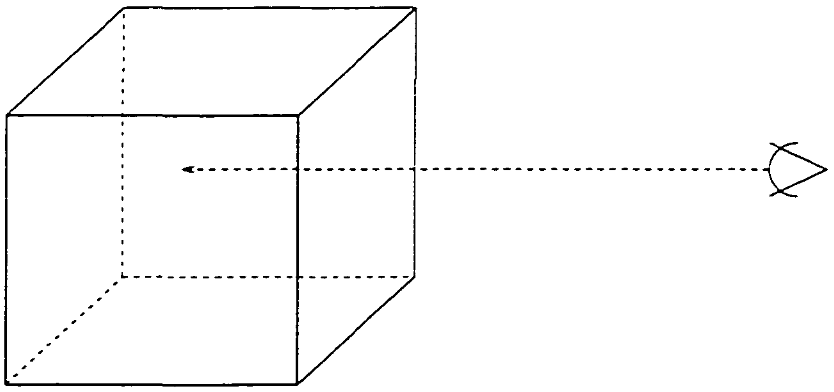

Figure 7.8: *Volume rendering on a single processor*

Figure 7.8 represents a volume being rendered on a single processor. This shows what happens with a ray tracing a single pixel within the image. Using the opacity map, there may come a point during the trace where the pixel becomes completely opaque. Nothing beyond this point in the volume will affect the final image, and thus the trace can be terminated. The point at which a ray terminates varies from pixel to pixel, with some tracing through the entire volume, and others tracing only a short distance. In the example shown in Figure 7.8, the point at which the trace is terminated falls just past the halfway point into the volume.

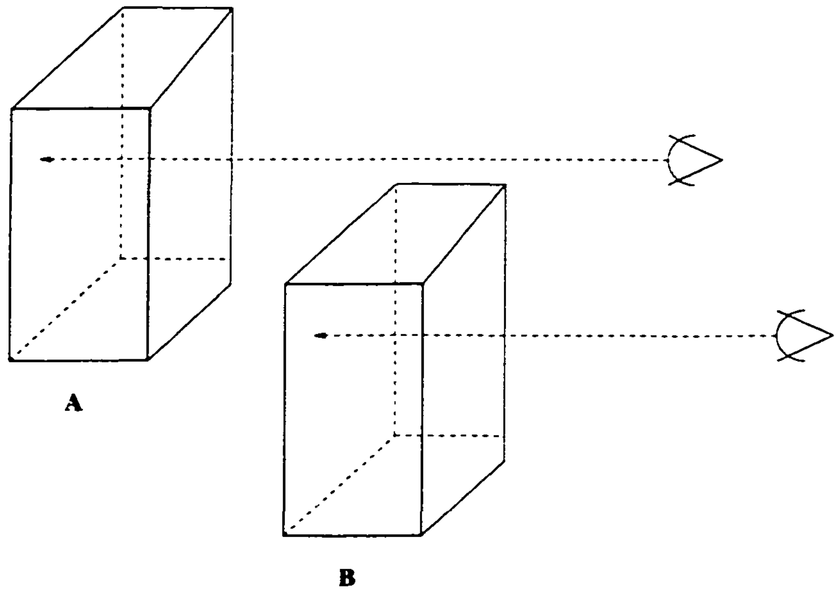

Figure 7.9: *Volume rendering on two processor*

Figure 7.9 represents the same rendering being performed on two processors. In this example, the volume is divided along the x-axis, with one processor rendering based on sub-volume A, and the other processor rendering based on sub-volume B. In this situation, the processor rendering sub-volume A has no knowledge of what is happening with sub-volume B. This means that it cannot use the opacity attained by the processor rendering sub-volume B in its decision on when to terminate tracing the ray. In this example, each processor traces the ray through its entire sub-volume. This is the equivalent amount of work as would have been done if the single processor case had traced the ray through the entire volume, which was not the case as stated above. In fact, since we stated that the single processor case terminated the trace just past the halfway point into the volume. the two processor case performs almost twice the amount of work.

This problem is exacerbated by multi-threading. Even when running on a single processor. multi-threading divides the volume into sub-volumes. with each thread being responsible for rendering based upon only its sub-volume. This causes the multi-threaded version to sometimes trace a ray through portions of the volume that are unnecessary. in the same way as multi-processing does. Additionally. when multi-threading on multiple processors. portions of rays are traced that need not be traced in the non-threaded multi-processor version.

These issues are orthogonal in that they have opposite effects. The cache effect tends to improve performance both when multi-threading and when running on multiple processors. Conversely. the opacity issue tends to increase the work being performed in both the multi-threaded and multiple processor cases. Which issue affects the total execution time the most depends on the conditions. The higher the opacity of the image. the worse the multi-threaded and parallel executions are going to perform. The smaller the cache. the more the cache effect is going to improve performance by multi-threading and parallelizing the execution.

Table 7.10 and Figure 7.10 show the results for the tests using the opacity map that made the image almost completely transparent. This not only makes for a very balanced parallel execution, it eliminates the performance degradation associated with opacity. That is, each ray will trace through the entire volume, never being truncated early. This means the same amount of work is performed regardless of the number of processors used or the number of threads used. This leaves cache effect as the only issue affecting execution times. As we can see from the numbers for non-threaded executions, cache effect has a dramatic effect, producing super-linear speedup. Each time we increase the number of processors, we see greater cache effect.

When we turn our attention to the multi-threaded numbers, we see better performance from the multi-threaded executions than we saw with the non-threaded executions. This is again due to cache effect. By breaking up the volume into smaller sub-volumes, the multi-threaded executions have better cache locality, and thus perform better. We again see super-linear speedup in the multi-threaded case running on multiple processors, due to yet smaller sub-volumes.

This makes it difficult to judge the overhead associated with multi-threading, as we could get the same results by changing the non-threaded version. We therefore concentrate on the overhead associated with enabling load balancing. What we see is a slight performance hit when load balancing is enabled. This will be further explained below.

Table 7.11 and Figure 7.10 show the execution times for rendering the same volume, but using an opacity map that represents a more average case situation. By looking at the

	Number of processors			
Implementation	1	2	4	8
Sequential non-threaded	308.0	—	—	—
MPI based non-threaded	308.0	144.6	68.1	33.9
Multi-threaded with no load balancing	258.4	128.5	63.1	31.7
Multi-threaded with load balancing	258.3	129.7	64.6	33.4

Table 7.10: *Execution time (in seconds) for volume rendering (Opacity map A)*

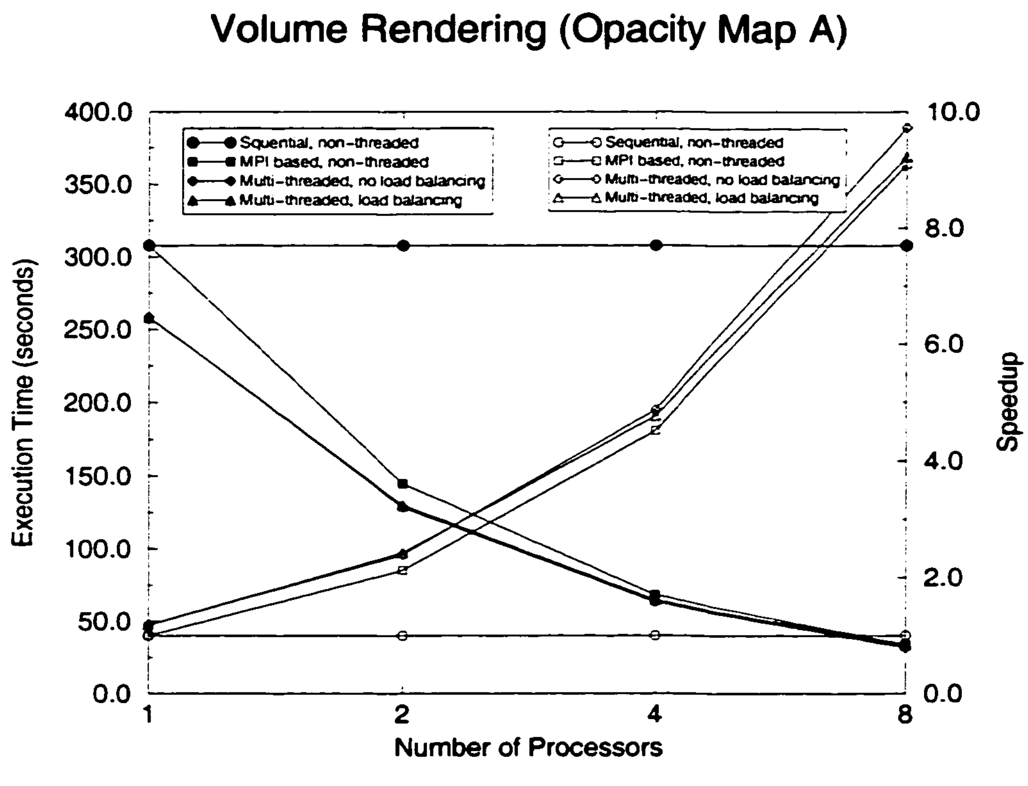

Figure 7.10: *Execution time and speedup for volume rendering (Opacity map A)*

execution times for the non-threaded case. we see very good speedup. This may lead us to believe that the load is fairly balanced across the processors. However, as we discussed above, the cache effects should cause super-linear speedup for a balanced load distribution. In this example, the performance degradation associated with opacity outweighs the benefits achieved from better cache locality. This can be seen clearly when we compare the sequential time to the execution time for the multi-threaded implementation on a single processor. The fact is. this example does not have a good load balance. and significant performance improvement should be available via load balancing.

When we compare the numbers for the non-threaded parallel version with the multi-threaded version with load balancing disabled, we see the multi-threaded version has longer execution times. This is due to the fact that the multi-threaded version has significantly more work to perform, as discussed above. This shows that the performance gains achieved by better cache locality are far outweighed by the increased work required due to the opacity effect.

It is important to point out that this is a product of the parallel algorithm, and not a shortcoming of multi-threading. Since the algorithm itself will not scale well in terms of the number of processors, it does not perform well in a multi-threaded environment.

In this case, it is difficult to evaluate the efficiency of multi-threading. However, we can still study the effects of enabling load balancing. We see from this comparison, that modest

	Number of processors			
Implementation	1	2	4	8
Sequential non-threaded	364.3	—	—	—
MPI based non-threaded	364.3	185.0	115.9	62.3
Multi-threaded with no load balancing	402.2	221.8	117.5	69.0
Multi-threaded with load balancing	402.8	216.1	109.5	58.0

Table 7.11: *Execution time (in seconds) for volume rendering (Opacity map B)*

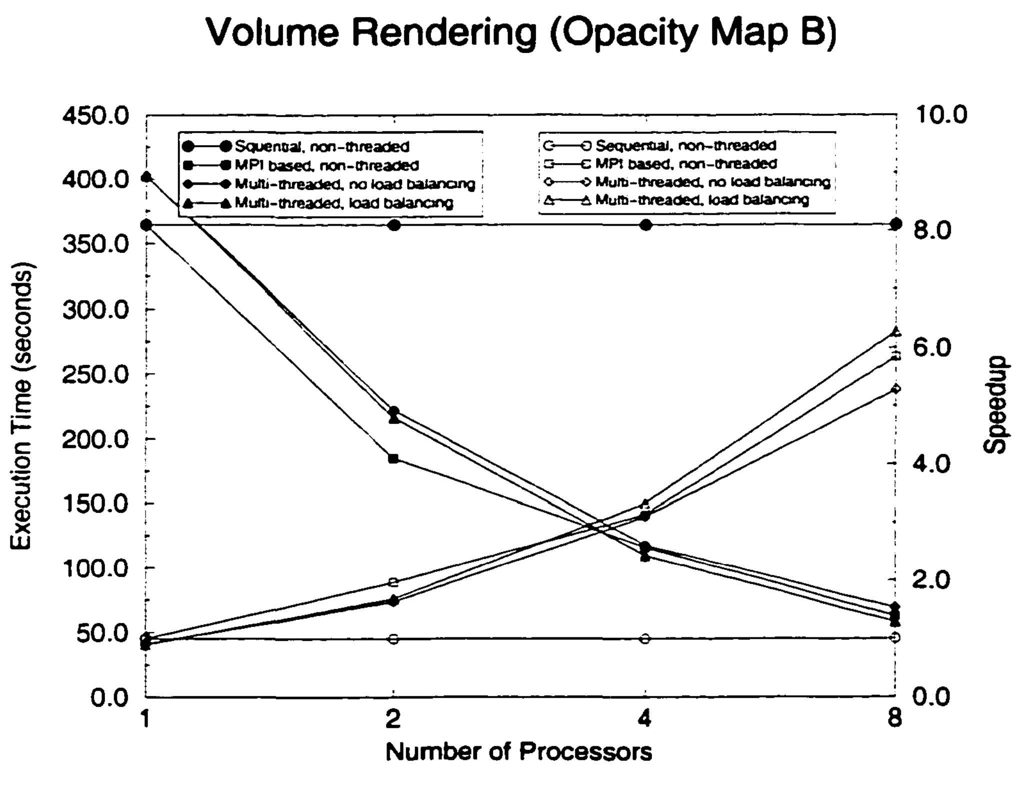

Figure 7.11: *Execution time and speedup for volume rendering (Opacity map B)*

performance gain can be achieved from enabling load balancing. However, studying the results led us to believe that there was a small amount of necessary overhead. The way the load balancing is being done in this example, all threads have the same load attributed to them throughout their life. This was similar to the previous examples where there was no good way to estimate how much computation was left for a thread.

The issue is that some threads that are very nearly done with their work may be migrated. The problem is, often these threads need less time to complete their computation than it takes to migrate them to another processor. This means that once the entire computation is nearly finished, we have threads migrating between processors when it would be faster for them to simply complete their computation. This accounts for the slight performance degradation we saw when enabling migration for opacity map A (Table 7.10).

In these volume rendering examples, however, we can estimate the amount of work remaining by the number of rows left to be scanned. The algorithm works by computing a sub-space of the image field for the final image. That is, it computes which parts of the final image will be affected by each sub-volume. It then scans each row of pixels within this sub-image. Thus, we know how many pixels remain that need to be traced. We used this information and set a thread's load to be equal to the number of rows it must scan. We then decrement the thread's load after each row has been scanned. This allows us to alter the load balancing policy to never migrate a thread with a load less than some value. We re-ran our multi-threaded executions with this new load balancing policy setting it up to never migrate a thread with a load of less than 5.

	Number of processors		
Opacity Map	2	4	8
A	128.4(129.7)	63.1(64.6)	31.8(33.4)
B	213.7(216.1)	108.6(109.5)	57.1(58.0)

Table 7.12: *Execution time (in seconds) for multi-threaded volume rendering*

Table 7.12 shows the execution times for the multi-threaded implementation with load balancing enabled and load for threads taken into account. The numbers in parentheses are the numbers for the old load balancing policy, taken from Tables 7.10 and 7.11.

We see from this table that this new load balancing policy slightly improves the execution times. In fact, if we look back to Table 7.10, we see that the execution times with this new load balancing policy are about identical to the times we saw with load balancing disabled. This suggests that there is no performance gain available for executions using opacity map *A*. This is what we would expect since it is nearly perfectly balanced. It also suggests that, though the performance gains are minimal, performance can be affected by the load balancing policy in use.

This chapter has presented the overhead associated with multi-threading as well as the overhead associated with thread migration. Furthermore, it has presented performance results for a number of test applications under varying conditions. These test were run sequentially, in parallel, and multi-threaded with and without load balancing.

Chapter 8

Conclusions

This dissertation can generally be divided into two parts. The first part is the design and implementation of a distributed light-weight threads system that supports point-to-point communication. The second part is the design and implementation of a thread migration mechanism, and the use of said mechanism for the purpose of dynamic load balancing.

This chapter explores what we have learned through both of these phases of work. We offer our conclusions regarding the effectiveness of using threads in a distributed memory environment, and our conclusions regarding the use of thread migration as a mechanism for dynamic load balancing. Finally, we discuss what we feel are the contributions of this work, and future work that may be pursued in the area of thread migration and load balancing.

1 Distributed Light-weight Threads

When we first started the work that led to this dissertation, the use of light-weight threads was just emerging as a model for parallel computation. However, this use was limited to shared-memory multi-processing. We noticed the lack of research on the use of light-weight threads by the distributed memory, multi-processing community, and decided it would be a worthwhile undertaking. Since that time, the use of light-weight threads in a distributed memory environment has received increased interest. In fact, some significant developments have come out of this interest. Today there are a number of packages that support this programming model.

However, these packages tend to be focused on specific problems. None of these existing packages support the wide range of programming models for which Chant is targeted. Moreover, since they are geared towards specific problems, they do not offer the generic solution provided through Chant.

We have created a light-weight threads package that runs in a distributed memory environment while supporting a wide range of functionality. This package supports remote thread operations, point-to-point communication, remote service requests, collective operations, and relative indexing. Chant also supports explicit message passing, explicit data sharing, or a combination of the two, as its programming model.

We have shown in Chapter 4. Section 4, that Chant introduces minimal overhead in the test cases studied. The majority of the overhead can be attributed to effects of supporting thread migration and load balancing. Thus we have shown that the base case of Chant (without migration support) is an efficient means for distributed memory multi-processing. We have, therefore, demonstrated that light-weight threads running in a distributed memory environment is a viable programming model, even when the threads package supports such a wide range of functionality.

2 Thread Migration and Load Balancing

Light-weight thread migration has received increased attention in recent years. However, the few packages that have actually been implemented have serious limitations. These limitations include, but are not limited to, redundant memory allocation, poor support for user-level pointers, and unacceptably high costs for accessing heap data.

Chant supports thread migration in a way that is unique. Chant is implemented totally as a runtime system, requiring no compiler support. Additionally, Chant performs migration in a manner that maintains all user-level pointers, at the cost of user registration of all pointers, and provides for continued point-to-point communication capabilities between migrant threads. Furthermore, Chant accomplishes this in a way that keeps overhead at a manageable level.

We have shown that the time it takes to migrate a thread across processor boundaries is linearly proportional to the size of the thread's stack and private heap, and that the additional overhead is within an acceptable range. This additional overhead includes the time it takes to forward messages, and the time necessary for registering and releasing pointers.

The use of thread migration for the purpose of load balancing is mentioned in almost all papers on thread migration. However, very little work has been done to study the feasibility of this approach. While Chrisochoides has proposed the use of thread migration as a means for achieving load balancing in Parallel Adaptive PDEs [13], he has not yet finished a working implementation. In fact, the last we knew he was no longer working on this problem.

The PM^2 [38, 42] project has published performance numbers for a single test case, a hand-written Gaussian elimination. While they show reasonable speedup for a very unbalanced distribution, they make no comparisons for relatively balanced distributions. Therefore we have no indication of the overhead associated with their implementation. Also, this is a very specialized package, only intended to be used for data parallel programming.

The work on load balancing with Active Threads [29] is also inconclusive. For starters, this package runs on a cluster of SMPs, which is clearly different that the work described in this dissertation. More importantly, however, as discussed earlier (Chapter 3, Section 12) the only performance numbers are based on starting the entire computation on a single SMP, and then migrating threads to the other SMPs within the cluster. This gives no indication of the effectiveness of this load balancing, since there is no static distribution across all the SMPs in the cluster against which to compare.

We have developed a load balancing system that is both simple and complex. That is, we have provided a default load balancing function that can be used with very little input from the user. At the same time, we have provided a complete load balancing API that allows a user to customize the load balancing layer, from simple changes, to complex changes. This provides a general purpose load balancing layer, which has been shown to perform well as is, but which can be manipulated to meet specific needs of individual applications.

We have tested this load balancing layer and arrived at three major conclusions. The first conclusion is that for executions that are already well-balanced across the available processors, our system adds minimal overhead. This is important, since for many applications it is not known before the actual execution how much imbalance may be present. A system that adds significant overhead would not perform well for well-balanced executions.

Secondly, we have seen that we can attain significant performance improvement for severely unbalanced executions. This performance improvement comes at a very low cost, due to the simple API we provide, which makes it easy for users to implement their code using our system.

Finally, we have seen that even for moderately unbalanced executions, we can achieve reasonable performance gain. This shows that the costs associated with thread migration, and with the decision making process of the load balancing layer, can easily be outweighed by the performance gained from the load redistribution.

3 Analysis

We feel that we have made significant contributions to the distributed memory multi-processing community. We have shown that multi-threading in a distributed memory environment can be useful, not only for ease of programming, but also for improved performance via thread migration. We have demonstrated that thread migration is a viable means of load redistribution and that significant performance gains can be attained. We have shown this to be true for a variety of applications, including branch and bound (TSP), divide and conquer (adaptive quadrature), and irregular scientific computation (volume rendering).

It is our belief that, as long as a technology is useful and or interesting, work on that topic is never finished. The same holds here. We have shown that thread migration is both interesting and useful. For that reason, this is by no means a finished work. We sincerely hope this research continues for many years to come, and that some day thread migration is so efficient and well understood that it is a common means of performing load redistribution.

With that in mind, it would be negligent of us not to point out some of the areas of this work that require further investigation. One of these areas is in message forwarding. While the current implementation performs reasonably well, we believe significant improvements are possible. There may indeed be a better way of both keeping track of messages as well ensuring that they arrive on the same processor as the target thread. This would be much easier if we were not interested in true point-to-point message passing, but we truly believe that point-to-point message passing is important, and the task is that much more difficult.

Another area that can use some more work is in keeping track of pointers. The current implementation works extremely well when there is not a lot of dynamic memory allocation and deallocation, but for problems like the Traveling Salesman Problem, significant time is spent in the register-release functions. An alternative would be indirect access to heap data, but for many applications this adds too much overhead. Therefore, other alternatives must be studied.

Finally, there is the issue of preemptive vs. non-preemptive threads. The current implementation uses a non-preemptive threads package. This means threads only block when the user explicitly yields the processor, or the runtime system blocks a thread for things such as message passing. This causes load balancing to be much coarser-grained than we would like. The use of preemptive threads would allow for much finer-grained load balancing, which would make this system attractive to a wider range of applications. However, converting Chant to a preemptive threads system would be a major undertaking. This is because there were many assumptions made during the implementation that would not hold for a preemptive package. Still, a preemptive package may offer significant advantages and this is definitely worth investigating.

These are only some of the areas that may offer future challenges. There surely are many other ares of interest for thread migration. As multi-processing research continues and processing speeds continue to increase, we must explore software techniques in an attempt to exploit new hardware technologies as the become available. Thread migration is one of these software techniques, and deserves continuing attention.

Appendix A

Load Balancing API

This appendix provides a summary of the load balancing layer of the Chant run-time system, as described in Chapter 6.

1 Lower Level Load Balancing Routines

The routines in the lower sub-layer of the load balancing layer (see Figure 6.1) are described in the following Subsection.

1.1 Control

The following routines are used to control the way load balancing works in a particular execution.

reg_lb_function (func_t func)

This function registers the user defined load balancing function *func*, which will be called by the system when load balancing is to be performed. If this function is not called, the default load balancing mechanism will be used.

reg_get_global_redistribution (func_t func)

This function registers the user defined function *func*, which can be called by the system or any user supplied function in place of the default implementation of the *get_global_redistribution* routine, as described later in this appendix. If this function is not called, the default implementation of *get_global_redistribution* will be used.

reg_get_local_redistribution (func_t func)

This function registers the user defined function *func*, which can be called by the system or any user supplied function in place of the default implementation of the *get_local_redistribution* routine, as described later in this appendix. If this function is not called, the default implementation of *get_local_redistribution* will be used.

reg_get_request_list (func_t func)

This function registers the user defined function *func*, which can be called by the system or any user supplied function in place of the default implementation of the *get_request_list* routine, as described later in this appendix. If this function is not called, the default implementation of *get_request_list* will be used.

reg_get_thread_redistribution (func_t func)

This function registers the user defined function *func*, which can be called by the system or any user supplied function in place of the default implementation of the *get_thread_redistribution* routine, as described later in this appendix. If this function is not called, the default implementation of *get_thread_redistribution* will be used.

reg_get_spec_redistribution (func_t func)

This function registers the user defined function *func*, which can be called by the system or any user supplied function in place of the default implementation of the *get_spec_redistribution* routine, as described later in this appendix. If this function is not called, the default implementation of *get_spec_redistribution* will be used.

begin_load_balancing (int upper, int lower, freq_t frequency)

This function enables load balancing. *upper*, *lower*, and *frequency* are all input parameters, where *upper* indicates the upper threshold, *lower* is the lower threshold used by either the default or user supplied load balancing routines. and *frequency* indicates the frequency of load balancing attempts. These values are used as described in Chapter 6. Section 1.1.

end_load_balancing ()

This function disables load balancing.

*define_neighborhood (int num. int *procs)*

This function defines the load balancing domain as described in Chapter 6. Section 1.1. Both *num* and *procs* are input parameters. where *num* is the number of processors in the load balancing domain. and *procs* is an array specifying the processors in the domain.

*get_neighborhood (int *num. int *procs)*

This function retrieves the current load balancing domain of the calling processor. Both *num* and *procs* are out parameters. On return. *num* is the number of processors in the load balancing domain. and *procs* is an array of length *num* specifying the processors in the domain.

1.2 Thread Characteristics

The following routines are used to retrieve thread characteristics.

get_my_load ()

This function returns the current load of the calling thread.

alter_load (int amt)

This function changes the load of the calling thread by *amt*.

get_thread_load (chanter_t t)

This function returns the load associated with the thread referred to by *t*.

get_local_thread_loads (int loads)*

This function retrieves the loads associated with all the threads on the run queue. The function returns the number of threads on the run queue and *loads* is an array containing the loads of each thread on the run queue.

*get_my_migratability (int *mig)*

This function returns, in *mig*, the migratability of the calling thread, as discussed in Chapter 6, Section 1.2.

get_migratability (chanter_t t)

This function returns the migratability, as discussed in Chapter 6, Section 1.2, of the thread referred to by *t*.

get_all_migratability (int migs)*

This function returns, in the array *migs*, as discussed in Chapter 6, Section 1.2, the migratability of all the threads on the run queue. Upon return, *migs[i]* is the migratability of the *ith* thread on the run queue.

set_migratability (int mig)

This function sets the migratability, as discussed in Chapter 6, Section 1.2, of the calling thread to *mig*.

*get_thread_total_comms (int *sends)*

This function sums up the total number of messages sent from each thread on the run queue. Upon return, *sends[i]* is the total number of messages sent by the thread in the *ith* position on the run queue.

get_thread_comms_proc (int proc, int sends[])

This function returns, in the array *sends*, the number of messages sent from each thread on the run queue to processor *proc*. Upon return, *sends[i]* is the number of messages sent from the *ith* thread on the run queue to processor *proc*.

*get_thread_comms (int flag, int *sends[])*

This function returns, in the array of pointers *sends*, the number of messages sent from each thread on the run queue to each processor in either the load balancing domain (*flag* = DOM) or the application (*flag* = ALL). If *flag* is ALL, upon return, *sends[i][j]* is the number of messages sent from the *ith* thread on the run queue to processor *j*.

If *flag* is DOM. upon return, *sends[i][j]* is the number of messages sent from the *ith* thread on the run queue to the *jth* processor in the load balancing domain. *sends[0][j]* are unused slots in the array.

get_spec_thread_comms (chanter_t t, int comm)*

This function returns, in array *comm*, the number of messages sent by thread *t* to each processor.

1.3 State Information

This section discusses routines used for gathering global state information.

get_local_load ()

This function returns the load of the local processor

*get_total_comms (int flag, int *sends)*

This function returns, in the array *sends*, the communication history of the local processor. If *flag* equals *DOM*, it returns the number of messages sent to each processor in the load balancing domain. If *flag* equals *ALL*, it returns the number of messages sent to each processor in the entire computation.

get_remote_loads (int flag, int loads)*

This function returns, in the array *loads*, the load of processors within the computation. If *flag* equals *DOM*, it returns the load of each processor in the load balancing domain. If *flag* equals *ALL*, it returns the load of each processor in the entire computation.

get_remote_comms (int flag, int sends[])*

This function returns, in the array *sends*, the communication histories of the processors within the computation. If *flag* equals *DOM*, it returns the communication histories of each processor in the load balancing domain. If *flag* equals *ALL*, it returns the communication histories of each processor in the entire computation.

1.4 Queue Manipulation

get_num_runq_threads ()

This function returns the number of threads on the local run queue.

get_first_thread ()

This function removes the first thread from the run queue, and returns a pointer to the Chant thread.

get_thread (int i)

This function removes the *ith* thread from the run queue, and returns a pointer to the Chant thread. If *i* is 0, it returns *nil*. If *i* is greater than the number of threads on the run queue, this function will wrap around the run queue and remove the appropriate thread. Upon return, the thread that had been in the *i + 1st* position on the run queue will be at the head of the run queue, with all previous threads in the same order, but at the end of the run queue.

put_thread (chanter_t t)

This function puts the thread referenced by *t* on the end of the run queue.

1.5 System Manipulation

synch_for_load_balancing ()

This routine synchronizes all the processors that are part of the calling processor's load balancing domain.

release_synch ()

This routine releases the processors that were synchronized with a corresponding call to the function *synch_for_load_balancing()*. Calling this routine without a corresponding synchronization is erroneous, and the results are undefined.

2 Load Balancing Directives

These routines are used to determine load redistributions that should be carried out (decision making routines), and to actually carry out the load redistributions (redistribution routines).

2.1 Decision Making Routines

*ret[][] get_global_redistribution (int flag, int *loads)*

This routine takes as input the load of each processor either in the calling processor's load balancing domain (*flag* = *DOM*), or the entire computation (*flag* = *ALL*). It uses this input to determine how much work each processor should move where. It returns a two-dimensional array, *ret*, where *ret[i,j]* indicates that processor i should move *ret[i,j]* work to processor j. i and j are processor ids relative to the entire computation (*flag* = *ALL*) or relative to the load balancing domain (*flag* = *DOM*).

The default implementation attempts to redistribute the work evenly, as described in Chapter 6, Section 2.

*get_local_redistribution (int flag, int *procs, int *remote_loads)*

This routine is called by an overloaded processor and is used to determine how much work should be sent to each processor in the computation (*flag* = *ALL*) or in the calling processor's load balancing domain (*flag* = *DOM*). *remote_loads* is an input array that provides the loads associated with the processors in the entire computation (*flag* = *ALL*) or in the load balancing domain *flag* = *DOM*). Upon return, *procs* contains the amount of work that should be moved to each participating processor. This array would be used as input to *get_thread_redistribution* as described below.

The default implementation selects the least loaded processor from *remote_loads* and indicates that half the calling processor's work should be moved to that processor.

*get_request_list (int flag, int *loads, int *remote_loads)*

This routine is called by an under-loaded processor and is used to determine how much work should be requested from each processor in the computation (*flag* = *ALL*) or in the calling processor's load balancing domain (*flag* = *DOM*). *remote_loads* is an input array that provides the loads associated with the processors in the entire computation (*flag* = *ALL*) or in the load balancing domain *flag* = *DOM*). Upon return, *procs* contains the amount of work that should be requested from each participating processor.

The default implementation selects the most heavily loaded processor, and indicates that half that processor's work should be moved to the calling processor.

*get_thread_redistribution (int *num, int *threads, int *procs, int *loads)*

This routine determines which threads from the calling processor are to be migrated to which processors in the load balancing domain of the calling processor (*flag* = *DOM*) or to which processors in the entire computation (*flag* = *ALL*). *loads* is an input array that states *loads[i]* work should be migrated to processor i (*flag* = *ALL*) or the *ith* processor in the load balancing domain (*flag* = *DOM*). Upon return, *num* contains the number of threads that must be migrated, *threads* is an array in increasing order that holds the displacement from the front of the run queue of the threads that must be migrated, and *procs* is an array indicating the processors to which the threads should be migrated.

The default implementation traverses the run queue, selecting migratable threads to be migrated to the destination processors. It selects threads for the first destination processor until enough work has been selected to satisfy that request and then continues with the remaining destination processors.

*get_spec_redistribution (int *num, int *threads, int proc, int load)*

This routine determines which threads from the calling processor are to be migrated to a specific processor. The input parameter *load* indicates how much work should be moved to specific processor *proc*. Upon return, *num* contains the number of threads that must be migrated, and *threads* is an array in increasing order that indicates the displacement from the front of the run queue of the threads that must be migrated.

The default implementation traverses the run queue, selecting migratable threads to migrate until enough load has been selected to satisfy the request.

2.2 Redistribution Routines

redistribute (int proc, int loads)*

This routine instructs a remote processor to migrate work to the other processors in the computation. *loads* is an input array that indicates the amount of work *proc* should move to each processor in the computation.

Upon return there is no guarantee that the migrations have been completed. This routine is not used by the default load balancing system and is provided solely for use in customized implementations. Care should be taken with its use.

*send_work (int num, int *threads, int proc)*

This routine instructs the calling processor to migrate *num* threads to processor *proc*. *threads* is an array in increasing order, indicating the displacement from the front of the run queue of the threads that are to be migrated.

Upon return, all indicated threads have been migrated.

*scatter_work (int num, int *threads, int *procs)*

This routine instructs the calling processor to migrate *num* threads to the processors indicated by *procs*. *threads* is an array in increasing order,, indicating the displacement from the front of the run queue of the threads that are to be migrated. The thread at displacement *threads[i]* is to be migrated to processor *procs[i]*.

Upon return, all indicated threads have been migrated.

get_work (int proc, int load)

This routine requests that *load* amount of work be migrated from processor *proc* to the calling processor.

Upon return, any migration to be performed has been completed.

*gather_work (int *loads)*

This routine requests work from remote processors. *loads* is an array indicating the amount of work to be requested from each processor in the computation.

Upon return, any migration to be performed has been completed.

*migrate_threads (int num, chanter_t *t, int proc)*

This routine instructs the system to migrate *num* threads to processor *proc*. *t* is an array containing pointers to the actual threads to be migrated. The threads referenced by *t* should **NOT** reside on the run queue. That is, the threads referenced by *t* should have been previously removed from the run queue using the routines described in Chapter 6, Section 1.4.

Upon return, all indicated threads have been migrated.

Bibliography

[1] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[2] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 125–134, May 1990.

[3] John K. Bennett, John B. Carter, and Willy Zwaenpoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, pages 168–176, March 1990.

[4] Raoul Bhoedjang, Tim Rühl, Rutger Hofman, Koen Langendoen, Henri Bal, and Frans Kaashoek. Panda: A portable platform to support parallel programming languages. In *Symposium on Experiences with Distributed and Multiprocessor Systems IV*, pages 213–226, San Diego, CA, September 1993.

[5] Andrew D. Birrell. An introduction to programming with threads. Technical Report 35, Digital Equiptment Corporation, January 1989.

[6] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86, Portland, OR, October 1986.

[7] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1), January 1987.

[8] E.A. Brewer, C.N. Dellarocas, A. Colbrook, and W.E. Weihl. Proteus: A high-performance parallel architecture simulator. Technical Report MIT/LCS/TR-516, MIT, September 1991.

[9] Jeremy Casas, Ravi Konuru, Steve W. Otto, Robert Prouty, and Jonathan Walpole. Adaptive load migration systems for PVM. In *Proceedings of Supercomputing*, pages 390–399, Washington D.C., November 1994. ACM/IEEE.

[10] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object oriented programming notation. In *Research Directions in Object Oriented Programming*. MIT Press, 1993.

[11] K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional parallel programming. In *Proceedings of the Fifth International Workshop on Parallel Languages and Compilers*, 1993.

[12] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield. The Amber system: Parallel programming on a network of multipocessors. In *ACM Symposium on Operating System Principles*, December 1989.

[13] Nikos P. Chrisochoides. Multithreaded model for dynamic load balancing parallel adaptive PDE computations. Technical Report CTC95TR221, Cornell University, October 1995.

[14] Nikos P. Chrisochoides. Multithreaded model for dynamic load balancing parallel adaptive PDE computations. Technical Report 95-83, Institute for Computer Applications in Science and Engineering, February 1996.

[15] E.C. Cooper and R.P. Draves. C Threads. Technical Report CMU-CS-88-154, Carnegie Mellon University, February 1988.

[16] Intel Corporation. *Paragon OSF/1 User's Guide*. Beaverton, OR, April 1979.

[17] Derek L. Eager, Edward D. Lazowska, and John Jahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, May 1986.

[18] Bryan Ford, Mike Hibler, and Jay Lepreau. Notes on thread models in Mach 3.0. Technical Report UUCS-93-012, Department of Computer Science, University of Utah, April 1993.

[19] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the Winter 1994 USENIX Conference*, January 1994.

[20] Message Passing Interface Forum. *Document for a Standard Message Passing Interface*. version 1.1 edition, June 1994. http://www.mcs.anl.gov/mpi/.

[21] Ian Foster, Carl Kesselman, Robert Olsen, and Steve Tuecke. Fortran M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 25(1), February 1995.

[22] Ian Foster, Carl Kesselman, Robert Olson, and Steven Tuecke. Nexus: An interoperability layer for parallel and distributed computer systems. Technical Report Version 1.3, Argonne National Labs, December 1993.

[23] Ian Foster, Carl Kesselman, and Steven Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[24] R. J. Fowler. Decentralized object finding using forwarding addresses. Technical Report 85-12-1, University of Washington, December 1985.

[25] A. Geist, A. Beguelin, and et. al. *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing*. ACM Press, 1994.

[26] M. Haines. On designing lightweight threads for substrate software. In *Proceedings of the Annual Technical Conference on UNIX and Advanced Computing Systems*, Anaheim, California, January 1997. USENIX.

[27] Matthew Haines, David Cronk, and Piyush Mehrotra. On the design of Chant: A talking threads package. In *Proceedings of Supercomputing*, pages 350–359, Washington D.C., November 1994. ACM/IEEE.

[28] Matthew Haines, Piyush Mehrotra, and David Cronk. Ropes: Support for collective operations among distributed threads. Technical Report 95-36, Institute for Computer Applications in Science and Engineering, November 1994.

[29] Michael Holtkamp. Thread migration with Active Threads. Technical Report TR-97-038, International Computer Science Institute, September 1997.

[30] Wilson C Hsieh, Paul Wang, and William E Weihl. Computation migration: Enhancing locality for distributed-memory parallel systems. In *Symposium on Principles and Practice of Parallel Programming*, 1993.

[31] IEEE. *Threads Extension for Portable Operating Systems (Draft 7)*, February 1992.

[32] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[33] Reinhard Luling and Burkhard Monien. A dynamic distributed load balancing algorithm with provabale good performance. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures*, 1993.

[34] Reinhard Luling, Burkhard Monien, and F. Ramme. A study on dynamic load balancing algorithms. Technical Report TR-001-92, Paderborn Center for Parallel Computing, June 1992.

[35] Mamoru Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.

[36] Edward Mascarenhas and Vernon Rego. Ariadne: Architecture of a portable thread system supporting mobile processes. Technical Report CSD-TR-95-017, Purdue University, March 1995.

[37] N. Melab, N. Devesa, M.P. Lecouffe, and B. Toursel. An adaptive load balancing algorithm with a multithreaded implementation. In *Eleventh International Conference On Systems Engineering (ICSE'96)*, Las Vegas, Nevada, July 1996.

[38] R. Namyst and J. F. Mehaut. PM^2: Parallel Multithreaded Machine. In *Proceedings of Parco '95*, Gent, Belgium, September 1995.

[39] R.M. Needham. *Distributed Sytems*. ACM Press, 1989.

[40] Shashank S. Nemawarkar and Guang R Gao. Measurement and modeling of earthmanna multithreaded architecture. Technical Report ACAPS96, School of Computer Science, McGill University, July 1995.

[41] Steve W Otto. Processor virtualization and migration for PVM. In *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, pages 66–75, Townsend, TN, May 1994. SIAM.

[42] C. Perez and R. Namyst. On the compilation of data-parallel languages on a distributed memory multithreaded environment with thread migration. Technical Report RR97-20, IP, July 1997.

[43] James Pinakis. Remote thread execution. In *Proceedings of the 16th Australian Computer Science Conference*, pages 489–500, Brisbane, Australia, February 1993.

[44] J. Sang, F. Knop, V. Rego, J.K. Lee, and C.T. King. The Xthreads Library: Design, implementation, and application. In *Proceedings of the COMPSAC*. 1993.

[45] Janche Sang and Vernon Rego. Efficient implementation of thread migration on distributed memory multiprocessors. Technical Report CSD-TR-93-065, Purdue University, October 1993.

[46] Neelakantan Sundaresan and Linda Lee. An object-oriented thread model for parallel numerical applications. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, pages 291–308, Sunriver, OR, April 1994.

[47] Dijkstra E. W., Feijen W. H. J., and Van Gasteren A. J. M. Derivation of a termination detection algorithm for distributed comnputations. *Information Processing Letters*, 16:217–219, June 1983.

[48] W.E. Weihl. E. Brewer, A. Colbrook, C. Dellarocas, W. Hsieh, A. Joseph, C. Waldspurger, and P. Wang. Prelude: A system for portable parallel software. Technical Report MIT/LCS/TR-519, MIT, October 1991.

[49] C. Xu, R. Luling, B. Monien, and F.C.M. Lau. An analytical comparison of nearest neighbours algorithms for load balancing in parallel computers. In *Proceedings of the 9th International Parallel Processing Symposium*, 1995.

[50] Honbo Zhou and Al Geist. Lpvm: A step towards multithread pvm. *Journal of Parallel and Distributed Computing*, July 1995.

VITA

David Charles Cronk

Born in Poughkeepsie, New York, September 30, 1966. Graduated from Franklin Delano Roosevelt High School in Hyde Park, New York, June 1984. B.S., Mathematics, Hope College, 1988. M.S., Computer Science, Marist College, 1992.

In September 1993, the author entered the College of William and Mary as a graduate student in Computer Science. The author worked as a student researcher at the Institute for Computer Applications in Science and Engineering from September 1993, through September 1998. The author has been working as a Member Technical Staff at Lucent Technologies as part of the Inferno Operating System venture since October 1998.