RICE UNIVERSITY

OpenMP on Networks of Workstations

by

Honghui Lu

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

__________________________
Dr. Willy Zwaenepoel, Chairman
Professor
Electrical and Computer Engineering
Computer Science

__________________________
Dr. Alan L. Cox
Associate Professor
Computer Science

__________________________
Dr. Edward W. Knightly
Assistant Professor
Electrical and Computer Engineering

Houston, Texas
February, 2001
OpenMP on Networks of Workstations

Honghui Lu

Abstract

The OpenMP Application Programming Interface (API) is an emerging standard for parallel programming on shared memory architectures. Networks of workstations (NOWs) are attractive parallel programming platforms because of their good price/performance ratio, as well as their flexibility and their potential to scale. This work is the first to extend the support for OpenMP to networks of workstations. The design is based on integrating the compiler and the run-time system. In the combined system, the run-time library remains the basic vehicle for implementing shared memory, while the compiler performs optimization rather than implementation.

The integrated approach can effectively optimize irregular applications, for which an exact compile-time analysis is not possible. One optimization aggregates messages for irregular applications based on the data access information provided by the compiler. Another optimization improves the scalability of the system by computation replication and multicast.

The integrated system also simplifies the run-time implementation. In the implementation of OpenMP on networks of shared-memory multiprocessors, the compile-time information greatly reduces the number of changes required to the run-time system in order to exploit the intra-node hardware shared memory. In another implementation that allows OpenMP programs to change the number of computing nodes during the execution, the OpenMP semantics provides convenient points for efficient adaptation.
Acknowledgments

I would like to thank my advisor, Willy Zwaenepoel, for his guidance and encouragement over the years. My thanks also go to Alan Cox, for lending his technical expertise whenever asked. Together, their vision and insights are instrumental to the completion of this thesis. I would like also to thank Edward Knightly for serving on my committee, and providing invaluable advice on this manuscript. My thanks go to Iva Jean Jorgensen and Kimberly M'Carver for proof reading this manuscript.

During the research toward this thesis, I cooperated with many people on various projects: Sandhya Dwarkadas, Charlie Hu, Eyle de Lara, and Ram Rajamony at Rice; and Thomas Gross and Alex Scherer at ETH Zürich. Without their cooperation, I would not have been able to complete this work by now. I wish to thank all of them for their invaluable help.

I thank all the past and current members of the Systems group at Rice University, for the great working environment they provide. In addition, my friends: Cristiana, Ram, Johnny, Zhenghua, Bo, Qing, Karthick, Mohit, Mike and others have made my life here very enjoyable. I am glad I had their company.

Last, but certainly not the least, I give my thanks to my family. I owe my parents and grandparents immense gratitude, for enabling me to be what I am today. My thanks also go to Weimin, for being a helpful colleague, a pleasant friend and a caring husband.
Contents

Abstract ii
Acknowledgments iii
List of Tables viii
List of Illustrations x

1 Introduction 1
1.1 OpenMP ............................... 1
1.2 OpenMP on Networks of Workstations .................... 1
1.3 Improving Performance ................................ 2
1.4 Increasing Flexibility ................................ 4
1.5 Thesis and Contributions ............................... 4
1.6 Dissertation Overview ................................ 6

2 OpenMP and TreadMarks 9
2.1 The OpenMP API ................................ 9
2.2 TreadMarks ................................. 12
2.2.1 Release Consistency and Multiple-Writer Protocol ........ 13
2.2.2 A Lazy Invalidate Implementation of Release Consistency ... 14
2.2.3 Modifications for OpenMP ........................ 17
2.3 The OpenMP to TreadMarks Translator ...................... 17
2.4 An Example .................................. 19
2.5 Summary .................................... 20

3 Applications 23
3.1 Jacobi ........................................ 23
3.2 SOR ........................................ 24
3.3 MGS ........................................ 24
3.4 3D-FFT ..................................... 24
3.5 LU .......................................... 25
3.6 Water ....................................... 26
3.7 TSP ........................................ 26
3.8 Moldyn ..................................... 27
3.9 NBF ........................................ 27
3.10 Barnes-Hut ................................. 28
3.11 ILINK ..................................... 29
3.12 Summary .................................. 29

4 Message Aggregation for Irregular Applications 31
4.1 Run-time and Compiler Optimizations ............. 34
  4.1.1 Example .................................. 34
  4.1.2 Augmented Run-Time System ................ 36
  4.1.3 Compiler Analysis .......................... 39
4.2 CHAOS ..................................... 41
4.3 Experimental Evaluation ........................ 42
  4.3.1 Moldyn .................................. 43
  4.3.2 NBF .................................... 46
4.4 Related Work ................................ 48
4.5 Summary .................................... 49

5 Improving Scalability 51
5.1 Contention .................................. 53
5.2 Design ...................................... 55
5.3 Implementation .............................. 57
6 Networks of Shared Memory Multiprocessors

6.1 Modifications to TreadMarks .......................... 76
6.2 Modifications to Translator ........................... 77
6.3 Results .................................................. 78
   6.3.1 OpenMP/original versus OpenMP/thread .......... 80
   6.3.2 OpenMP versus MPI ............................... 84
6.4 Related Work ........................................... 85
6.5 Summary ............................................... 86

7 Transparent Adaptive Parallelism ........................ 88

7.1 Transparent adaptation ................................. 89
7.2 Implementation of the adaptive system ............... 91
    7.2.1 Join events .................................. 93
    7.2.2 Normal leave events .......................... 93
    7.2.3 Urgent leave events ........................... 94
    7.2.4 Fault tolerance ............................... 94
    7.2.5 Current limitations ........................... 95
7.3 Performance ............................................ 95
    7.3.1 Key overall results ........................... 96
    7.3.2 Micro analysis of adaptation costs ............. 100
8 Conclusions and Future Work 109
  8.1 Conclusions ........................................... 109
  8.2 Future Work ...........................................

Bibliography 112
### Tables

4.1 Moldyn - 8 processor results. The interaction list is updated at varying intervals. ................................. 43

4.2 NBF Kernel - 8 processor results. ................................. 47

5.1 Barnes-Hut. 32 nodes (Ideal speedup 28.6). ......................... 65

5.2 Barnes-Hut communications. 32 nodes. ................................. 66

5.3 Ilink. 32 nodes (Ideal speedup 18.9). ................................. 69

5.4 Ilink communications. 32 nodes. ................................. 69

6.1 Problem size and sequential execution time. ................................. 78

6.2 Amount of data and number of messages on four four-processor nodes. ................................. 80

6.3 Number of mprotects, page faults, and diffs on four SMP nodes. 1 – use only one processor per node. 4 – use four processors per node. ................................. 82

6.4 mprotect counts of OpenMP/thread with 16 processes on four four-processor nodes. One thread per process. ................................. 83

6.5 Average mprotect time (usec) on four four-processor nodes. ................................. 84

7.1 Problem size and sequential time of applications. ................................. 96

7.2 Execution times and network traffic on the non-adaptive system and the adaptive system with no adapt events. Network traffic is identical on both systems. ................................. 97

7.3 Normal leave, normal join, and migration times (sec). ................................. 98
7.4 Costs for one leave (8 → 7) ........................................ 102
7.5 Costs for two leaves (8 → 6) ........................................ 102
7.6 Costs for one join (7 → 8), two joins (6 → 8) and a leave/join (6 → 6) 103
7.7 3D-FFT: Costs for leaves and joins ................................. 103
Illustrations

2.1 A simple OpenMP program. ........................................... 21
2.2 TreadMarks program generated by the translator. ............... 22

4.1 Moldyn - main program and the subroutine ComputeForces ....... 35
4.2 Transformations for the Subroutine ComputeForces in Moldyn ...... 35
4.3 Augmented Run-Time Interface for Indirect Accesses ............... 37

5.1 Contention. ..................................................................... 53
5.2 Race condition. ................................................................. 59

6.1 Speedups on four four-processor nodes. ................................. 79

7.1 (Normal) join (a), normal leave (b), and urgent leave (c). ....... 90
7.2 Effect of thread id of leaving node on the amount of data moved at adaptation time, with a simple block distribution of the data among threads. The shaded areas indicate the data to be moved. Up to 50% of the data space is moved if node 7 leaves (a), only up to 30% if node 3 leaves. ................................................................. 98
7.3 Execution times for different intervals between adapt events. .... 99
Chapter 1

Introduction

1.1 OpenMP

The OpenMP Application Programming Interface (API) is an emerging standard for parallel programming on shared memory multiprocessors. It defines a set of program directives and a run-time support library that augment standard C, C++ and Fortran [Ope97, Ope98].

OpenMP specifically addresses the needs of scientific programming. It supports parallel programming at a level higher than POSIX threads, another shared memory API. In contrast to MPI [Mes94], a message passing API, OpenMP is much easier to program. Specifically, OpenMP facilitates an incremental approach to the parallelization of sequential programs: the programmer can add a parallelization directive to one loop or subroutine of the program at a time. Because of its ease of use, OpenMP seems to be attracting wide-spread support among both hardware and software vendors and among application developers, and it is the first successful effort to standardize shared memory programming directives (see http://www.openmp.org).

1.2 OpenMP on Networks of Workstations

OpenMP previously existed only for shared memory architectures, putting it at a disadvantage compared to the message passing interface MPI, which runs on both shared memory and distributed memory architectures.

Distributed memory architectures, especially networks of workstations (NOW), are increasingly attractive for parallel computing. Networks of workstations consist
of commodity workstations connected by commonly available interconnects. They are potentially more cost effective, more scalable, and more flexible than shared memory multiprocessors. Commercially available multiprocessor machines usually contain a small number of processors. Large multiprocessors will always have a worse price/performance ratio than commodity machines since their high hardware development cost must be amortized over a lower sales volume. In fact, connecting small scale multiprocessor machines with the network is a cost effective approach to build large scale parallel machines. Workstation clusters are also more flexible than multiprocessor machines. In the cluster environment, nodes can be easily added and removed after the deployment of the system, and they do not have to be homogeneous. The flexibility provides a clear upgrade path as faster processors and interconnects are produced. In addition, with the penetration and the speed of the Internet, any set of machines can corporate to perform parallel computation, and they need not be located within the local area network.

Because memory is inherently distributed across the nodes, the prevailing programming interface on NOWs is the message passing interface MPI. In MPI, the application programmer must take care of the distribution of the data, the partition of the computation, and the communication of the data shared across different nodes. The complexity of the message passing paradigm severely limits the appeal of NOWs as a platform for parallel scientific computing.

In general, our goal is to provide an efficient, scalable and flexible implementation of OpenMP on networks of workstations. This will not only improve the portability of OpenMP and further its acceptance as a parallel programming standard, but also will make parallel computing available to a much larger audience.

### 1.3 Improving Performance

To support shared memory interfaces on distributed memory machines, the system must handle the communication in software, and automatically generate the under-
lying message passing code. Because of the relatively high latency of commodity networks, the number of messages processed by a node is crucial to the performance of the system. There are traditionally two completely different approaches to build shared memory interfaces on distributed shared memory machines, which are either pure run-time or pure compile-time, and both have their own performance problems.

A software distributed shared memory (SDSM) system (e.g., [LH89]) provides a shared memory abstraction on a distributed memory machine using purely run-time mechanisms. During execution, a SDSM system detects shared memory accesses, handles faults by fetching the missing data, and caches data for future reference. Such a system can handle any kind of data access pattern with reasonable efficiency. However, the on-demand data fetching causes extra messages and consistency actions, increasing overhead and resulting in reduced performance compared to hand-tuned message passing. This problem is exacerbated with the increase of the number of nodes in the system.

The pure compiler approach directly generates the message passing code at compile-time (e.g., [App93, HKT92]). The compiler analyzes memory access patterns to generate message passing code, which is then optimized to aggregate communication or use multicast to reduce overhead. For programs with regular access patterns that can be precisely analyzed, these compile-time systems provide superior performance since they avoid the run-time overhead present with SDSM systems. However, when the access patterns cannot be analyzed precisely, the message passing code generated by the compiler becomes inefficient. In the case of irregular accesses, for example, a simplistic compiler approach would result in a broadcast of all data produced by a processor, causing large amounts of communication.

In improving performance, our goal is to efficiently support both regular and irregular programs on large scale clusters. Particularly, we want to optimize communication for irregular applications, which at present cannot be effectively optimized by either the pure run-time approach or the pure compile-time approach alone.
1.4 Increasing Flexibility

To be truly flexible, a NOW environment has to accommodate heterogeneous processing nodes, tolerate a variety of network latencies, and adapt to the changing cluster configuration on the fly. Among the many problems to be solved, we address support for networks of shared memory multiprocessors (SMP), and adaptation to a changing pool of computing nodes.

Since small multiprocessors are becoming commonplace, networks of shared memory multiprocessors are cost-effective for building large scale DSM systems. Therefore, it is important to build portable and efficient support for OpenMP on networks of SMPs. Although each vendor has a unique implementation of the shared memory hardware, most of them support the standard POSIX thread library, which provides efficient low-level shared memory programming within a multiprocessor. A portable and efficient OpenMP implementation would use POSIX thread within a node. The challenge of such a system is to seamlessly integrate the shared memory interface used within a node and the underlying message passing communication across nodes.

Unlike dedicated computing environments, individual nodes in a NOW become available or unavailable as the workstation owner goes away or returns. In addition, there is a higher possibility of node failure on large scale systems. To be truly useful, a parallel processing system for a NOW must be able to adapt to a continually changing pool of available nodes. Ideally, this adaptation should be transparent, allowing the user to program in a relatively standard way, without requiring any special-purpose code in the application. As a result, the system must make efficient decisions on when to adapt to the changing environment, and how to repartition the data and the computation after the adaptation.

1.5 Thesis and Contributions

This dissertation centers around the idea that by integrating the run-time system with the semantic information provided by the compiler, we can provide efficient
and flexible shared memory support for both regular and irregular applications on
distributed memory machines.

There are many ways to combine the benefits of SDSM systems with those of
the compiler-based approaches. We could build a system that is highly optimized
on regular applications for which the compiler analysis succeeds, and falls back to
the run-time approach for irregular applications, providing an acceptable but less
optimized performance. However, by closely integrating the run-time and the compile-
time systems, we can provide optimized performance for both regular and irregular
applications.

This dissertation makes two major contributions toward our thesis.

I. The design of a general infrastructure for efficient integration of the
run-time and the compile-time systems.

In the combined system, the run-time library remains the basic vehicle for imple-
menting shared memory, while the compiler performs optimization rather than im-
plementation. Instead of generating a message passing program directly, the compiler
generates a shared memory program augmented with run-time calls that describe the
program behavior. By informing the run-time system of future program behavior,
these calls allow the run-time system to optimize communication.

II. The design and implementation of different mechanisms for integrating
the compile-time system and the run-time system, and using them to solve
a variety of problems to improve the performance and the availability of
OpenMP on NOWs.

We explore the vast design space of the compiler/run-time integration mechanisms.
The run-time system needs different information from the compiler in order to solve
different problems. In our experiments, the compiler provides information includ-
ing data access patterns, parallelization status, and whether a variable is shared or
not. By using this information, the run-time system either optimizes communication,
changes the number of available nodes in a parallel machine, or hides the difference in memory models.

1.6 Dissertation Overview

Chapter 2 explains the basic implementation of OpenMP on NOWs. In order to support OpenMP execution on NOWs, our compiler targets a software distributed shared memory system (SDSM). The compiler translates OpenMP to fork-join parallel programs that run on the SDSM system. In the fork-join model of parallel execution, the sequential code sections are executed by a single thread, called the master thread. The parallel code sections are executed by all threads, including the master thread. Chapter 2 describes the OpenMP API, the target SDSM system, and the translation of OpenMP to the underlying SDSM programs. Chapter 3 introduces the applications used in this dissertation, and their OpenMP implementations.

Chapter 4 presents an integrated compiler/run-time approach to aggregate messages for irregular applications. It involves, in addition to the DSM runtime system, a simple compiler front-end that generates data access information, enabling the runtime system to efficiently precompute the data that will be accessed by each thread. These data are then requested, prior to the actual accesses, in a single message exchange with each thread from which data is needed.

Chapter 5 studies methods to scale OpenMP to a large number of processing nodes. Our method targets the bottleneck caused by sequential sections in the OpenMP programs. Parallel programs usually have sequential sections for initializing data, and for parts of the code that are either too complicated or too expensive to be performed in parallel. In the shared memory paradigm, the sequential sections are executed by one thread, which is usually the master thread in OpenMP. Results of the sequential computation are accessed by all threads later in the parallel computation, causing all threads to contend for updates from the master. This contention gets worse with the increase of the number of processing nodes and becomes a major
performance bottleneck on large scale systems. Our solution takes advantage of the fork-join programming model of OpenMP to automatically replicate the sequential execution on all nodes. The replicated sequential execution allows each thread to apply modification to the local copy of the shared data. This completely eliminates the need to propagate changes from the master thread, and eliminates contention at its source. During the replicated sequential execution, we take advantage of the fact that every thread executes the same code to make efficient use of multicast. This optimization does not rely on exact compile-time data analysis, and is completely integrated into the SDSM runtime. As a result, even irregular pointer based applications can benefit from our optimizations.

Chapter 6 reports an implementation of OpenMP on networks of shared-memory multiprocessors. This system enables the programmer to rely on a single, standard, shared-memory API for parallelization within a multiprocessor and between multiprocessors. In contrast to previous SDSM systems for SMPs, our system uses POSIX threads for parallelism within an SMP node. We achieve transparency by exploiting the OpenMP variable annotations, which specify whether or not a variable is shared among the threads. This approach greatly simplifies the changes required to the SDSM in order to exploit the intra-node hardware shared memory.

Chapter 7 presents a system that allows OpenMP programs to execute on a network of workstations with a variable number of nodes. The ability to adapt to a variable number of nodes allows a program to take advantage of additional nodes that become available after it starts execution, or to gracefully scale down when the number of available nodes is reduced. Two ideas underlie the efficiency of our design. First, we recognize that OpenMP programs exhibit convenient adaptation points during their execution, points at which the cost of adaptation can be much reduced. Second, by allowing a thread a certain grace period before it must leave a node, we ensure that most adaptations can occur at these adaptation points, and thus at low cost. Migration of a thread, a much more expensive method for providing adaptivity,
is used only as a back-up solution, when the thread cannot reach an adaptation point within the grace period. We demonstrate that the cost of adaptation is modest; the system allows a program to adapt at a moderate rate without much performance loss.
Chapter 2

OpenMP and TreadMarks

The OpenMP Application Program Interface (API) [Ope97, Ope98] supports shared memory parallel programming in C/C++ and Fortran programs. OpenMP allows a step-wise migration from a sequential program to a parallel one, independent of the availability of tools for automatic parallelization. Therefore, this approach to parallelization is highly popular among users. However, OpenMP previously existed only for shared memory architectures, putting it at a disadvantage compared to the message passing interface MPI, which runs on both shared memory and distributed memory machines. We implemented the first prototype of OpenMP on distributed memory machines, and in particular on a network of workstations (NOW) [LHZ98]. Such an implementation lends increased portability to OpenMP programs and thereby furthers its acceptance. We use the TreadMarks software distributed shared memory (DSM) system to implement a shared memory abstraction on a NOW, and target the compiler to the interface provided by that software DSM. This chapter explains the OpenMP Application Program Interface (API), the TreadMarks software distributed shared memory system, and the basic implementation of OpenMP on TreadMarks.

2.1 The OpenMP API

The OpenMP Application Program Interface (API) [Ope97, Ope98] specifies a collection of compiler directives, library functions, and environment variables that can be used to specify shared memory parallelism in C/C++ and Fortran programs. OpenMP is based on a fork-join model of parallel execution. The sequential code sections are executed by a single thread, called the master thread. The parallel
code sections are executed by all threads, including the master thread. OpenMP provides three kinds of directives: parallelism/work sharing, data environment, and synchronization directives. In C/C++, the directives are implemented as `#pragma` statements, and in Fortran 77/90 they are implemented as comments. In addition to the compiler directives, OpenMP also provides a small set of library functions and environment variables. We only explain the directives relevant to this thesis, and refer interested readers to the OpenMP standard [Ope97, Ope98] for the full specification.

The fundamental construct for expressing parallelism is the `parallel` directive. It defines a `parallel region` of the program that is executed by multiple threads. The parallel directive takes an optional `if` clause. When the `if` clause is present, the parallel construct is executed in parallel only when the `if` expression evaluates to a non-zero value. There is an implied barrier at the end of the parallel region, and only the master thread continues execution afterwards. Once started, the number of threads must remain constant while that parallel region is being executed. However, the number of threads can be changed either explicitly by the user or automatically by the runtime system from one parallel region to the other.

Work sharing constructs divide the computation among the threads within a parallel region. For example, the `for` directive divides the iterations of the associated loop among the threads so that each iteration is performed by a single thread. The `for` directive can take a `schedule` clause that specifies the assignment of the iterations to threads such as cyclic or block. There is an implicit barrier at the end of a work sharing construct unless a `nowait` clause is specified. OpenMP also defines shorthand forms for specifying a parallel region which contains a single work sharing directive. The `parallel for` directive is shorthand for a `parallel` region that contains a single `for` directive.

The data environment directives control the sharing of program variables that are defined outside of a parallel region.* They include the `threadprivate` directive

---

*The variables defined inside a parallel region are implicitly private.*
and a set of data scope attribute clauses. The data scope attribute clauses allow
a user to control the attribute of variables for the duration of the parallel region.
They appear at the beginning of a parallel region, immediately following the parallel
directives. The data scope attributes include shared, private, firstprivate,
lastprivate, and reduction. Each directive is followed by a list of variables. Variables
default to shared, which means they are shared among all threads in a parallel
region. A private variable has a separate copy per thread. Its value is undefined
upon entering or exiting a parallel region. In addition to having the same attributes
as private variables, firstprivate variables are initialized to the master’s value
at the time the parallel region is entered. The lastprivate clause also provides a
superset of the functionality provided by the private clause. When a lastprivate
clause appears on the directive that identifies a work-sharing construct, the value
of each lastprivate variable from the sequentially last iteration of the associated
loop is assigned to the variable’s original object. The reduction directive identifies
reduction variables, and the operations performed on them. According to the Version
1.0 of the standard, reduction variables must be scalar, but we extend the standard
to include arrays. In contrast to the data scope attribute clauses, the threadprivate
directive appears at file scope or name space scope, outside of any definition or decl-
aration. The threadprivate directive makes the named file scope or name space
scope variables specified in the list private to a thread, but their data persist between
parallel regions. The value of threadprivate variables is undefined when the number
of threads changes between parallel regions.

The synchronization directives include barrier, critical, and flush. A barrier
directive causes the thread to wait until all other threads in the parallel region have
reached this point. After the barrier, all threads are guaranteed to see all modifications
made before the barrier. A critical directive restricts access to the enclosed
code to only one thread at a time. When a thread enters a critical section, it is guar-
anteed to see all modifications made by all the threads that have entered the critical
section earlier. The flush directive specifies a “cross thread” sequence point in the program where all threads are guaranteed to have a consistent view of the variables named in the flush directive, or of all of the memory if no variables are specified.

The library functions allow the program to obtain the number of threads and the thread identifier, and to enable or disable the dynamic adjustment of the number of threads. These library functions set the corresponding environment variables at runtime.

We implement a subset of the OpenMP API for our experiments. This includes the parallel construct, the work sharing constructs, the combined parallel work sharing constructs, and the library functions. We support static block or cyclic partition of loops. We do not support nested parallel regions. All the data environment directives excluding lastprivate are implemented in our system, so are the critical and barrier synchronizations.

2.2 **TreadMarks**

TreadMarks [ACD+96] is a user-level software distributed shared memory (SDSM) system that runs on most Unix and Windows NT-based systems. It provides a global shared address space on top of physically distributed memories. The parallel threads synchronize via primitives similar to those used in hardware shared-memory machines: barriers and locks. A Tmk::barrier(i) call makes the calling thread wait at the barrier until all other threads have reached this barrier. A Tmk::lock_acquire(i) call acquires a lock for the calling thread, and Tmk::lock_release(i) releases it. No thread can acquire a lock while another thread is holding it. In C, the program has to call the Tmk::malloc routine to allocate shared variables in the shared heap. In Fortran, the shared data are placed in a common block loaded in a standard location.

TreadMarks relies on the operating system’s virtual memory page protection mechanism to detect accesses to the shared pages, and the threads communicate via software messaging such as UDP on a local area network. Because sending messages
between two threads is very expensive, TreadMarks makes great effort to minimize synchronization messages [KDCZ94].

2.2.1 Release Consistency and Multiple-Writer Protocol

TreadMarks features a release consistent (RC) [DKCZ93, AH93] shared memory model. In the RC model, shared memory accesses are categorized either as ordinary or as synchronization accesses, with the latter category further divided into acquire and release accesses. RC requires ordinary shared memory updates by a thread $p$ to become visible to another thread $q$ only when a subsequent release by $p$ becomes visible to $q$ via some chain of mutual synchronizations. In practice, this model allows a thread to buffer multiple writes to shared data in its private memory until the release. In TreadMarks, Tmk_lock_acquire(i) is modeled as an acquire, and Tmk_lock_release(i) is modeled as a release. Tmk_barrier(i) is modeled as a release followed by an acquire, where each thread releases at barrier arrival, and acquires at barrier departure.

False sharing is another cause of frequent communication. False sharing occurs when two or more threads access different variables within the same page, with at least one of the accesses being a write. TreadMarks uses a multiple-writer protocol to address this problem. With the multiple-writer protocol, two or more threads can simultaneously modify their own copy of the shared page. Their modifications are merged at the next synchronization operation in accordance with the definition of RC, thereby reducing the effect of false sharing. In order to distinguish changes made by different threads, instead of sending the whole page, only the modified values are sent. Those modified values are called diffs, meaning the difference between the modified page and the page before the modification.
2.2.2 A Lazy Invalidate Implementation of Release Consistency

TreadMarks implements a lazy invalidate version of release consistency [KCZ92]. The propagation of modifications is postponed until the time of the acquire. The releaser is lazy: it does not propagate modifications to the shared pages. Furthermore, instead of sending new data to the acquirer, the releaser notifies the acquirer of which pages have been modified, causing the acquirer to invalidate its local copies of these pages. A thread incurs a page fault on the first access to an invalidated page, and gets the diffs for that page from previous releasers.

The acquiring thread determines which modifications it needs to see according to the definition of RC. To do so, the execution of each thread is divided into intervals. A new interval begins every time a thread executes a release or an acquire. Each thread has an interval index, which is incremented every time a new interval starts on this thread. Intervals on different threads are partially ordered [AH93]: (i) intervals on a single thread are totally ordered by program order, (ii) an interval on thread \( p \) precedes an interval on thread \( q \) if the interval of \( q \) begins with the acquire corresponding to the release that concluded the interval of \( p \), and (iii) an interval precedes another interval by transitive closure. With locks, the interval corresponding to the release of a lock directly precedes the interval beginning with the subsequent acquire to the same lock. With barriers, any interval corresponding to the barrier arrival precedes all intervals corresponding to the subsequent barrier departures. However, no ordering exists among the barrier arrivals, or among the barrier departures.

Each interval has a vector timestamp to record its knowledge of intervals in other threads that precede it. A timestamp contains an entry for each thread. For example, in the timestamp of the \( i \)th interval of thread \( p \), the entry for thread \( p \) is equal to \( i \). The entry for thread \( q \) other than \( p \) denotes the most recent interval of thread \( q \) that precedes interval \( i \) of thread \( p \) according to the partial order.

RC requires that before a thread \( p \) may continue past an acquire, the updates of all intervals preceding the current interval must be visible at \( p \). Therefore,
an acquire, \( p \) sends its current interval timestamp to the previous releaser \( q \). The releaser then compares the corresponding entries of both timestamps, and sends a message to \( p \) including \textit{write notices} for all intervals named in \( q \)'s current interval timestamp, but not in the timestamp it received from \( p \). Thread \( p \) computes a new vector timestamp according to the pair-wise maximum of its previous timestamp and the releaser's timestamp. A \textit{write notice} is an indication that a page has been modified in a particular interval.

\textbf{Optimizations for Diff and Interval Creation}

In order to capture the modifications to a shared page, it is initially write-protected. At the first write, a protection violation occurs. The DSM software makes a copy of the page (a \textit{twin}), and removes the write protection so that further writes to the page can occur without any DSM intervention. The twin and the current copy can later be compared to create a \textit{diff}, a runlength encoded record of the modifications to the page.

In TreadMarks, diffs are only created when a thread requests the modifications to a page or a write notice from another thread arrives for that page. In the latter case, it is essential to make a diff in order to distinguish the modifications made by the different threads. This \textit{lazy} diff creation is distinct from Munin's implementation of multiple-writer protocols, where at each release a diff is created for each modified page and propagated to all other copies of the page. The \textit{lazy} implementation of RC used by TreadMarks allows diff creation to be postponed until the modifications are requested. \textit{Lazy} diff creation results in a decrease in the number of diffs created and an attendant improvement in performance [KDCZ94].

Logically, a new interval begins at each release and acquire. In practice, interval creation can be postponed until we communicate with another thread, avoiding overhead if a lock is reacquired by the same thread. When a thread releases a lock to another thread or arrives at a barrier, a new interval is created containing a write no-
tice for each page that was twinned since the last remote synchronization operation. With lazy diff creation these pages remain writable until a diff request or a write notice arrives for that page. At that time, the actual diff is created, the page is read protected, and the twin is discarded. A subsequent write results in a write notice for the next interval.

**Synchronization**

Barrier arrivals are modeled as releases, and barrier departures are modeled as acquires. Barriers have a centralized manager. At a barrier arrival, each thread sends a release message to the manager, and waits for a departure message. The manager broadcasts a barrier departure message to all threads after all have arrived at the same barrier.

The two primitives for mutex locks are lock release and lock acquire. Each lock has a statically assigned manager. The manager records which thread has most recently requested the lock. All lock acquire requests are sent to the manager and, if necessary, forwarded by the manager to the thread that last requested the lock.

**Garbage Collection**

Garbage collection is necessary to reclaim the space used by write notice records, interval records, and diffs. During garbage collection, each thread validates its copy of every page that it has modified. All other pages, all interval records, all write notice records and all diffs are discarded. In addition, each thread updates the copyset for every page. If, after garbage collection, a thread accesses a page for which it does not have a copy, it requests a copy from a thread in the copyset.

The threads execute a barrier-like protocol, in which threads request and apply all diffs created by other threads for the pages they have modified themselves. Garbage collection is triggered when the amount of free space for consistency information drops below a threshold. An attempt is made to make garbage collection coincide with a
2.2.3 Modifications for OpenMP

To support OpenMP-like environments, recent versions of TreadMarks include the `Tmk_fork` and `Tmk_join` primitives, specifically tailored to the fork-join style of parallelism expected by OpenMP and most other shared memory compilers [AALT95]. For performance reasons, all threads are created at the start of a program’s execution. After the startup, the master thread executes the program while the slave threads are blocked inside the runtime system waiting for the master to issue a `Tmk_fork`. `Tmk_fork` is a one-to-all synchronization: the master thread sends messages to the waiting slave threads to wake them up. After the parallel execution, every thread calls `Tmk_join`. `Tmk_join` causes the master to wait for all threads to finish the parallel execution, and causes each slave to send a message to the master upon finishing the parallel region. After all messages are received, the master continues with the program and the slave threads are blocked waiting for the next `Tmk_fork`.

All variables default to shared in OpenMP. In TreadMarks, however, shared variables must be explicitly allocated on the shared heap. TreadMarks keeps the stack of each thread in private memory, and makes it inaccessible to other threads. To solve this problem, we introduce `Tmk_change_stack` to switch a thread’s stack to a region of memory allocated on the shared heap, so that it is visible to all threads. We will discuss how this primitive is used to implement the OpenMP data environment directives in the next section.

In addition, we also add the necessary functions in TreadMarks to support the OpenMP library functions.

2.3 The OpenMP to TreadMarks Translator

Our translator is based on the SUIF toolkit [AALT95]. The OpenMP to TreadMarks translation is relatively simple, because TreadMarks already provides a shared mem-
ory API on top of a network of computers. A large part of the translator deals with transforming the parallel regions to the fork-join format, and implementing the data environment directives. The OpenMP synchronization directives translate directly to TreadMarks synchronization operations.

To translate a sequential program annotated with parallel directives into a fork-join parallel program, the translator encapsulates each parallel region into a separate subroutine. This subroutine also includes code, generated by the compiler, that allows each thread to determine, based on its thread identifier, which portions of a parallel region it needs to execute. Variables defined outside the parallel region are redefined in the new subroutine. The private variables keep their original types, and the shared variables change to the reference types derived from their original types. At the time of the fork, the master passes a pointer to this subroutine to the slave threads. Pointers to shared variables and initial values of firstprivate variables are also copied into a structure and passed to the slaves at the fork.

Shared variables are implemented by allocating them on the shared heap. To make the address of shared variables available to the slaves, the master passes the address to the slaves at the fork. The translator gathers all shared global variables in a structure, and allocates the structure in shared memory. To implement shared automatic variables, _tik_change_stack_ is called by the master right after the startup, so that the master's stack is moved to the shared memory, and becomes accessible to the slave threads. The slaves' stack is still in private memory.

Private variables are implemented by redefining them in the parallel subroutine generated by the compiler, so that each thread accesses the private copy on their own stack. We implement firstprivate variables by passing their value from the master to the slaves at the fork. Two copies of each reduction variable are declared, with one being shared and another being private. During the parallel region, each thread accumulates local results in the private copy, and then combines its private results with the shared copy at the end of the parallel region. Threadprivate variables
are left alone during the translation, because all global variables are private to each thread in TreadMarks.

## 2.4 An Example

We conclude this chapter by showing a simple OpenMP program to demonstrate how OpenMP facilitates incremental parallelization, and how the translator handles OpenMP data environment directives. The interested reader can go to the official OpenMP website at www.openmp.org for extensive examples on OpenMP programming.

The program in Figure 2.1 tries to parallelize a single loop in the subroutine `init_A`. The loop adds a constant `C` to each element of array `temp` and creates the resulting array `A`. Because the parallel region only contains one loop, we parallelize it with the `parallel for` directive, and block partition is used to distribute the loop iterations. Both the global array `A` and the local array `temp` are shared. They are not listed in the data environment directives because `shared` is the default attribute for variables accessed inside the parallel region. The loop index variable `i` is `private`, and another variable `C` is `firstprivate`. The variable `C` is `firstprivate` so that we can pass the initial value of `C` from the sequential section to the parallel region. Notice that the directives are inserted right in front of the loop, without affecting the rest of the program. It is up to the translator to generate the full fledged parallel program.

The resulting TreadMarks program is shown in Figure 2.2. The translator defines additional global variables and subroutines in the output. Structure `Tmk_shared-global` contains all the shared global variables. Structure `fork arguments` contains arguments passed from the master to the slaves at fork. Finally, subroutine `init_A.sub1` encapsulates the parallel loop in `init_A`.

---

*The same effect can be achieved by declaring `C shared.*
The parallel subroutine \texttt{init.A_SUB1} redefines all variables accessed inside the parallel loop, including the global variable \texttt{A}. But the type of shared variables \texttt{A} and \texttt{temp} is changed from the array type to the pointer type of the corresponding array elements. The private variables \texttt{i} and \texttt{C} are simply redeclared in the subroutine. Two new local variables \texttt{upper_bound} and \texttt{lower_bound} are defined to hold the values of the loop bounds on each particular thread. The variables \texttt{proc_id} and \texttt{nprocs} are defined in the OpenMP runtime library. In \texttt{init.A_SUB1}, the compiler first inserts code to unpack the parameters passed from the master thread. Shared variables \texttt{A} and \texttt{temp} are passed by reference, and the \texttt{firstprivate} variable \texttt{C} is passed by value. After that, each thread calculates the lower and upper bounds of its chunk of the iteration space according to their \texttt{proc_id} and the number of threads. Finally, the loop is copied from the original subroutine, with the loop bounds replaced with the new values.

In the original subroutine \texttt{init.A}, the compiler first generates codes to pack the pointers or values into \texttt{fork_arguments}, and then calls \texttt{Tmk_fork} with the subroutine \texttt{init.A_SUB1} and the pointer to \texttt{fork_arguments} as arguments. The \texttt{Tmk_fork} is followed immediately by a \texttt{Tmk_join}.

In the main program, \texttt{Tmk_change_stack} is called at the beginning of the code. After that, the structure \texttt{Tmk_shared_global} is allocated in the shared memory.

### 2.5 Summary

This chapter explains the OpenMP API, the TreadMarks software distributed shared memory system, and the basic implementation of OpenMP on TreadMarks. Our approach relies on the runtime SDSM system to implement a shared memory interface on networks of workstations. The compiler simply generates the fork-join code, allocates shared memory, and converts OpenMP annotations to appropriate calls to the SDSM system. In the optimizations presented in the rest of the thesis, the runtime
#define N 1000
float A[N];

main() {
    ...
    init_A();
    ...
}

void init_A() {
    float temp[N];
    int i, C;

    C = 4;
    initialize(temp);

    #pragma omp parallel for schedule(block)
    #pragma omp private(i) firstprivate(C)
    for (i=0; i< N; i++)
        A[i] = temp[i]+C;
}

Figure 2.1: A simple OpenMP program.

remains the basic vehicle for implementing shared memory, and the compiler performs optimization rather than implementation.
#define N 1000
struct Tmk_shared_global {
    float A[N];
} *sharedGlobals;

struct Tmk_fork_arg {
    float *A, *temp;
    int C;
} fork_arguments;

extern int proc_id, nprocs;
void init_A_sub1(struct Tmk_fork_arg *arguments) {
    float *A, *temp;
    int i, C, upper_bound, lower_bound;

    A = arguments->A;
    temp = arguments->temp;
    C = arguments->C;

    lower_bound = (N*proc_id)/nprocs;
    upper_bound = (N*(proc_id+1))/nprocs-1;
    for (i=lower_bound; i<upper_bound; i++)
        A[i] = temp[i]+C;
}

main() {
    Tmk_change_stack();
    sharedGlobals = Tmk_malloc(sizeof(struct Tmk_shared_global));
        ... ...
    init_A();
        ... ...
}

void init_A() {
    float tmp[N];
    int i, C;

    C = 4;
    initialize(temp);

    fork_arguments.A = sharedGlobals->A;
    fork_arguments.temp = temp;
    fork_arguments.C = C;

    Tmk_fork(init_A_sub1, &fork_arguments);
    Tmk_join();
}

Figure 2.2: TreadMarks program generated by the translator.
Chapter 3

Applications

Our OpenMP implementation efficiently supports a large variety of applications on networks of workstations (NOW). Our studies in the thesis employ eleven applications, and each study chooses a subset of the applications according to their different characteristics. Among the applications, Jacobi, SOR and MGS are simple numerical computations; TSP solves the traveling salesman problem; 3-D FFT is from the NAS benchmark suite [BBL93]; and LU decomposition, Water, and Barnes-Hut are from the SPLASH-2 benchmarks [WOT+95]. We also include Moldyn from the CHAOS suite [MSH+95], NBF from the GROMOS benchmarks [GB88], and ILINK [CIS93, LLJO84], a widely used genetic linkage analysis program. This chapter describes these applications and their OpenMP implementations in detail.

3.1 Jacobi

Jacobi is an iterative method for solving partial differential equations. Its data is organized as a two-dimensional array. In the main computation, an entry in the array is assigned the value of the average of its four neighbors. The major data structures are a pair of two-dimensional arrays, one containing the data, the other used as a scratch array. There are two loops in each iteration, one updates the scratch array according to values in the main array, and the other copies the scratch array back to the main array. Both loops are parallelized with the parallel for directive. Both of the arrays are shared, and block distribution is applied to the loop iterations.
3.2 SOR

Red-Black Successive Over-Relaxation (SOR) is a method for solving partial differential equations by iterating over a two-dimensional array. The red and black elements in the array are interleaved so that each red element is surrounded by four black elements, and vice versa. In every iteration, each of the array elements is updated to the average of the element’s four nearest neighbors. All of the black elements are updated in a single loop, then all the red elements are updated in another loop. These data parallel loops are expressed in OpenMP using the `parallel for` directive. The data array is shared, and the loop iterations are partitioned into blocks.

3.3 MGS

Modified Gramm-Schmidt (MGS) computes an orthonormal basis for a set of N-dimensional vectors. During the i\textsuperscript{th} iteration, the algorithm first normalizes the i\textsuperscript{th} vector, then makes all vectors j > i orthogonal to vector i. The second part of each iteration is parallelized with `parallel for`, with a cyclic distribution of the loop iterations in order to reduce communication while balancing the load in each iteration. The two-dimensional array containing the vectors is shared in the parallel region.

3.4 3D-FFT

3-D FFT [BBL93] numerically solves a partial differential equation using three dimensional forward and inverse FFT’s. The program performs 1-D FFT along each dimension during each iteration. For example, assume the input array \( A \) has dimensions \( n_1 \times n_2 \times n_3 \), and is organized in row-major order. The 3-D FFT first performs an \( n_3 \)-point 1-D FFT on each of the \( n_1 \times n_2 \) complex vectors, and then it performs an \( n_2 \)-point 1-D FFT on each of the \( n_1 \times n_3 \) vectors. Finally, the resulting array is transposed into an \( n_2 \times n_3 \times n_1 \) complex array \( B \), and an \( n_1 \)-point 1-D FFT is applied
to each of the $n_2 \times n_3$ complex vectors.

In OpenMP, each iteration is encapsulated in a parallel region. Both arrays $A$ and $B$ are shared within the parallel region. The for directive is used to annotate the data parallel loops for each 1-D FFT and the transpose. The iterations of each loop are partitioned into blocks. The nowait clause is added to all parallel loops except the one immediately preceding the transpose. Because each thread applies a 1-D FFT to its own data, synchronization is unnecessary between two adjacent 1-D FFT loops.

### 3.5 LU

LU [WOT+95] factors a dense matrix into the product of a lower triangular and an upper triangular matrix. To exploit temporal locality on submatrix elements, the dense $n \times n$ matrix is divided into an $N \times N$ array of $B \times B$ blocks ($n = NB$). The blocks are distributed among threads in a 2D scatter decomposition, with each block being updated by the thread that owns it. Elements within a block are allocated contiguously to improve spatial locality. Furthermore, the blocks owned by the same thread are allocated consecutively in shared memory. The program consists of $N$ iterations. During the $i$th iteration, the program first factors the diagonal block $A_{ii}$, then updates all perimeter blocks in column $i$ and row $i$ using $A_{ii}$. Finally, the program updates interior blocks using corresponding perimeter blocks.

In OpenMP, the parallelism is expressed explicitly with the whole factorization enclosed in a parallel directive. Array $A$ and the blocks are shared within the parallel region. Because a block or cyclic partition of the loop iterations cannot guarantee that each thread updates its own blocks, the program defines a function to find the owner of each block based on the number of threads, and matches the owner with the thread identifier of each thread. The threads synchronize via barriers between the different phases of computation.
3.6 Water

Water [WOT+95] is a molecular dynamics simulation. The main data structure in Water is a one-dimensional array of molecules. During each time step, both intra- and inter-molecular forces are computed. Before calculating the forces, the program first predicts each molecule's displacement and its derivatives. Afterwards, the program corrects the predicted values based on forces computed, and calculates the boundary conditions, the kinetic energy, and the potential of each molecule.

In OpenMP, each step is enclosed in a parallel region. The molecule array is shared among the threads. The loops are annotated with the for directive, with block decomposition, and a nowait clause whenever possible. The evaluation of inter-molecular force involves accumulation of forces on both local and remote molecules. This is expressed with a reduction directive on the force arrays. The computation of the kinetic energy and the potential are also reductions on the corresponding variables. The bulk of the communication results from the synchronization that takes place during the inter-molecular force computation. During the rest of the program each thread mostly accesses its own molecules.

3.7 TSP

TSP solves the traveling salesman problem using a branch-and-bound algorithm. The major data structures are a pool of partially evaluated tours, a priority queue containing pointers to tours in the pool, a stack of pointers to unused tour elements in the pool, and the current shortest path. A thread repeatedly dequeues the most promising path from the priority queue, either extends it by one city and enqueues the new path, or takes the dequeued path and tries all permutations of the remaining cities.

In OpenMP, the main program is enclosed in a parallel region. All the major data structures are shared between the threads. Accesses to the priority queue and to the shortest path are synchronized using the critical directive.
3.8 Moldyn

Moldyn [MSH+95] is a molecular dynamics simulation. Its computational structure resembles the non-bonded force calculation in CHARMM [BBO+83], which is a well-known molecular dynamics method used to model macromolecular systems. Non-bonded forces are long-range interactions between each pair of molecules. CHARMM approximates the non-bonded calculation by ignoring all pairs which are beyond a certain cutoff radius. The cutoff approximation is achieved by maintaining an interaction list of all the pairs within the cutoff distance, and iterating over this list at each timestep. The interaction list is used as an indirection array to identify interacting partners. Since molecules change their spatial location every iteration, the interaction list must be periodically updated.

In order to improve spatial locality, the OpenMP version of Moldyn uses Recursive Coordinate Bisection (RCB) to assign the molecules to threads. Each thread constructs its interaction list independently, by examining its own molecules and half of the molecules behind it in the molecule list, in a wrap around fashion. The OpenMP program encloses each iteration in a parallel directive. The force array, the coordinate array, and the interaction list are shared among the threads. Within the parallel region, each thread finds its part of the molecules according to its thread identifier. The inter-molecular force computation is a reduction operation performed on the force array.

3.9 NBF

NBF [vH93] is the kernel of a molecular dynamics simulation which has been used previously as an example to demonstrate compiler generated message passing programs. Instead of keeping a list of pairs of interacting molecules like Moldyn, NBF keeps a list of interacting partners for each molecule. The lists of partners are concatenated together, with a per-molecule list pointing to the end of each molecule's partners in the partner list. For each molecule, the program goes through the list
of partners, and updates the forces on both a molecule and its partner based on the
distance between them. In our experiments, the partner list is static. Each molecule
has approximately the same number of partners distributed evenly in the memory.

The OpenMP program annotates the force computation loop with `parallel for`
Block decomposition is applied to the loop iterations. The force array, the coordinate
array, and the partner list are shared within the parallel region, and the update to
the force array is recognized as a reduction operation.

3.10 Barnes-Hut

Barnes-Hut [WOT+95] is an N-body simulation program using the hierarchical Barnes-
Hut method. A tree structure is used to represent the recursively decomposed sub-
domains (cells) of the three-dimensional physical domain containing all of the par-
ticles. The leaves of the tree correspond to the particles, and are contained in a
separate array.

Each iteration is divided into two steps. In the first step, a single thread reads
the particles and rebuilds the tree. The second step performs the force evaluation in
which all threads participate. First, the threads divide the particles by traversing the
tree in the Morton ordering (a linear ordering of the points in higher dimensions) of
the cells. Specifically, the i\textsuperscript{th} thread locates the i\textsuperscript{th} segment. The size of a segment is
weighted according to the workload recorded from the previous iteration. Then, each
of the threads performs the force evaluation for its particles, which involves a partial
traversal of the tree. During the force evaluation, a thread only modifies its own
particles. Finally, each thread updates the displacements of its particles according to
the forces applied to them.

In OpenMP, the second step is parallelized using the `parallel` construct, and
threads synchronize with the `barrier` directive before updating the displacements.
3.11 ILINK

ILINK [CIS93, LLJO84] is a widely used genetic linkage analysis program that locates specific disease genes on chromosomes. The input to ILINK consists of several family trees. The program traverses the family trees and visits each nuclear family. The main data structure in ILINK is a pool of genarrays. A genarray contains the probability of each genotype for an individual. Since the genarray is sparse, an index array of pointers to non-zero values in the genarray is associated with each one of them. A bank of genarrays large enough to accommodate the biggest nuclear family is allocated at the beginning of execution, and the same bank is reused for each nuclear family. When the computation moves to a new nuclear family, the pool of genarrays is reinitialized for each person in the current family. The computation either updates a parent’s genarray conditioned on the spouse and all children, or updates one child conditioned on both parents and all the other siblings.

We use the parallel algorithm described by Dwarkadas et al. [DSC+94]. Updates to each individual’s genarray are parallelized with the parallel directive. The bank of genarrays is shared among the threads. The master thread first examines the amount of work involved in the update, and decides to perform the update in parallel only if the amount of work exceeds a threshold. An if clause is used to express the conditional parallelization. A master thread then assigns the non-zero elements in the parent’s genarray to all threads in a cyclic fashion. After each thread has worked on its share of non-zero values and updated the genarray accordingly, the master thread sums up the contributions of each of the threads.

3.12 Summary

The applications can be classified according to either their access patterns or their parallelization patterns.

In terms of access patterns, the applications can be classified as either regular or irregular. Jacobi, SOR, MGS, 3D-FFT, LU, and Water are regular, because their
data access patterns can be fully analyzed at compile time. On the other hand, Moldyn, NBF, Barnes-Hut, ILINK, and TSP are *irregular*. Moldyn and NBF access data through indirection arrays, and Barnes-Hut, ILINK, and TSP are pointer based. Because the address of the data being accessed is not determined until runtime, it is very hard for the compiler to analyze their access pattern.

In terms of different forms of parallelism, TSP is task parallel, and all the other programs are data parallel. In *data parallel* code, all threads simultaneously perform the same operation on different parts of a data structure. We can express this kind of parallelism in OpenMP with the `parallel for` directive, or allow each thread to explicitly find its portion of the computation by their thread identifier, and synchronize with a *barrier* after the computation. In the task parallel TSP program, however, threads may perform different operations on different data structures at a certain time. The threads synchronize with *critical* in OpenMP. For the purpose of our studies, the key difference between data and task parallelism is that in data parallelism, there is a global synchronization point such as a barrier or the join operation where each thread waits until every thread has arrived at the same point. Task parallelism, on the contrary, only employs point-to-point synchronization.

Since compilers can do a good job at analyzing regular applications and generating message passing code, we focus on irregular applications in the two performance optimization studies, namely message aggregation in Chapter 4 and replicated sequential execution in Chapter 5. The study of networks of SMPs in Chapter 6 uses all kinds of applications. Finally, our automatic adaptation mechanism in Chapter 7 currently relies on global synchronization points in the program, and we include a mix of data parallel applications in that study.
Chapter 4

Message Aggregation for Irregular Applications

Parallel programming using a shared memory platform has the advantage of ease-of-use. In contrast to message passing, the user does not have to worry about data location or does not have to explicitly manage communication. Unfortunately, as parallel computers move away from the uniform memory access model to networks of workstations, this transparency of shared memory comes into question. Message passing programs, tuned to non-uniform memory access latencies, often produce better performance. Our goal is to develop a system that continues to provide the user with a transparent shared memory programming model, but underneath is capable of exploiting the hardware’s message passing capabilities.

A software distributed shared memory (SDSM) system (e.g., [LH89]) provides a shared memory abstraction on a distributed memory machine using purely run-time mechanisms. During execution, a SDSM system detects shared memory accesses, handles faults by fetching the missing data, and caches data for future reference. Such a system can handle any kind of data access pattern. However, the on-demand data fetching causes extra messages and consistency actions, increasing overhead and resulting in reduced performance compared to hand-tuned message passing [LDCZ95].

Research and commercial compilers for parallel computing on distributed memory machines have, to date, directly targeted the underlying message passing layer (e.g., [App93, HKT92]). The compiler analyzes memory access patterns to generate message passing code, which is then optimized to aggregate communication and to minimize data movement. For programs with regular access patterns that can be precisely analyzed, these compile-time systems provide superior performance since
they avoid the run-time overhead present with SDSM systems. However, when the access patterns cannot be analyzed precisely, the message passing code generated by the compiler becomes inefficient. In the case of irregular accesses, for example, of the form $A(index(i))$, a simplistic compiler approach could not identify which elements of array $A$ are being accessed, and would result in a broadcast of all data produced by a processor, causing large amounts of communication.

Our goal is to combine the benefits of SDSM systems with those of compiler-based approaches for generating code for distributed memory systems. Specifically, our optimization targets the indirection arrays of the form $A(index(i))$, for which an exact compile-time analysis is very complicated. In the combined system, the run-time library remains the basic vehicle for implementing shared memory, while the compiler performs optimization rather than implementation. Instead of generating a message passing program directly, the compiler generates a shared memory program augmented with run-time calls that describe the data access patterns. By informing the run-time system of future shared access patterns, these calls allow the run-time system to aggregate several data fetches into a single message.

Our approach involves, in addition to the DSM runtime system, a simple compiler front-end that generates data access information, enabling the run-time system to efficiently precompute the set of pages that will be accessed by each thread during the next iteration. These pages are then requested, prior to that iteration, in a single message exchange with each thread from which data is needed. In other words, our approach extends the base software DSM layer by enabling it to aggregate the communication of data for irregular programs. The compiler support required for our approach is very simple: it suffices to determine the indirection array, and the part of the indirection array being accessed by each thread. This is usually a regular section [CK88].

Inspector-executor methods [SBW91] have been proposed in compiler based systems as a way to efficiently execute irregular computations on distributed memory
machines. A separate loop, the *inspector*, precedes the actual computational loop (called the *executor*). The inspector loop determines the data read and written by the individual threads executing the computational loop. This information is then used to compute a *communication schedule*, moving the data from the producers to the consumers at the beginning and/or end of each loop. Communication aggregation is used to reduce the number of messages exchanged. In order to further reduce overhead, an attempt is made to execute the inspector loop only once for a large number of iterations of the executor loop. It has been argued that part or all of the above procedure can be automated by a compiler [vHKK+92]. The compiler analysis involved can, however, be quite complicated [AS95, DHSK95, vHK94].

This chapter presents our approach in detail. In order to gather experimental results, we use a modified version of TreadMarks [ACD+96] that supports prefetching and aggregation in the manner described above. Furthermore, we have augmented the ParaScope parallel programming environment [KMT93] to carry out the required compiler analysis. The compiler analysis and transformation are applied to the SUIF generated TreadMarks programs. We present performance results for two irregular applications, Moldyn and NBF. The results were obtained on an 8-processor IBM SP2 using base TreadMarks and TreadMarks with aggregation support. We compare these results to measurements of hand-coded inspector-executor versions of the same applications that use the CHAOS run-time library [DUSH94].

The outline of the rest of the chapter is as follows. Section 4.1 presents the run-time and compiler support for irregular applications. Section 4.2 provides a summary of CHAOS, the leading run-time system for support of irregular applications on message passing platforms. Section 4.3 presents the results of our evaluation of the shared memory run-time support, and compares the results to those from CHAOS. Section 4.4 provides a summary of related work. Finally, we conclude in Section 4.5.
4.1 Run-time and Compiler Optimizations

This section describes the enhancements to the TreadMarks run-time system as well as the compiler analysis necessary in order to aggregate messages for programs with irregular access patterns.

Our approach involves a simple compiler front-end that identifies the indirection array(s) to the run-time system. Specifically, the compiler performs a source-to-source transformation of the TreadMarks program, inserting calls to the (augmented) run-time DSM library before the indirect accesses. These calls identify the base addresses of the data arrays, and the sections of the indirection arrays accessed by a particular thread. The run-time system then uses this information to determine the set of shared pages that this thread accesses. The pages are requested in a single message exchange with each of the threads from which data is required.

In order to avoid having to recompute the set of pages accessed on every iteration, the run-time system subsequently write-protects the shared pages containing the indirection array. If no memory protection violation occurs for these pages, then the same set of pages are requested in the next iteration. Otherwise, the indirection array has been changed, and the set needs to be recomputed.

4.1.1 Example

We first illustrate our approach with an example using the Moldyn program (see also Section 3.8). Moldyn \cite{MSH+95} is a molecular dynamics simulation that computes non-bonded forces between pairs of molecules. Moldyn approximates the non-bonded calculation by ignoring all pairs which are beyond a certain cutoff radius. The cutoff approximation is achieved by maintaining an interaction list of all the pairs within the cutoff distance, and iterating over this list at each timestep. The interaction list is used as an indirection array to identify interacting partners. Since molecules change their spatial location every iteration, the interaction list must beperiodically updated.
PROGRAM MOLDYN

DO step = 1, NSTEPS
  !$OMP PARALLEL
  !$OMP SHARED(x, interaction_list)
  !$OMP REDUCTION(+, forces, n)
    IF (mod(step, UPDATE_INTERVAL).eq. 0) then
      call build_interaction_list()
      n1 = interaction_list(1, i)
      n2 = interaction_list(2, i)
    ENDIF
    ... ...
    call ComputeForces(x, force, interaction_list)
    forces(n1) = forces(n1) + force
    forces(n2) = forces(n2) - force
  ENDIF
  ... ...
  END
END

SUBROUTINE ComputeForces(x, force, interaction_list)

  Validate(i, INDIRECT,x,interaction_list[1:2,1:my_num_interactions],READ,1)

DO i = 1, my_num_interactions
  n1 = interaction_list(1, i)
  n2 = interaction_list(2, i)
  force = f(x(n1) - x(n2))
  forces(n1) = forces(n1) + force
  forces(n2) = forces(n2) - force
ENDDO
END

Figure 4.1: Moldyn - main program and the subroutine ComputeForces

Figure 4.2: Transformations for the Subroutine ComputeForces in Moldyn
Figure 4.1 illustrates the OpenMP version of Moldyn, highlighting the main program where subroutines build\_interaction\_list and ComputeForces are called. The subroutine ComputeForces is where the indirect accesses occur, and it is also depicted in detail. The OpenMP program encloses each iteration in a parallel directive. The force array, the coordinate array, and the interaction list are shared among the threads. The inter-molecular force computation is a reduction operation performed on the force array. Although not shown in the figure, the force array is initialized to zeros at the beginning of each iteration. As a result, a thread does not have to bring it up to date before the initialization.

The message aggregation transformations only affect the force computation subroutine, where the indirect accesses take place. Figure 4.2 shows the program transformations applied to ComputeForces. Because this subroutine does not have any OpenMP directives, the unoptimized TreadMarks code is identical to the OpenMP code. The compiler optimization inserts a Validate at the start of Compute\_Forces. The Validate initializes the data structures for the fetch. Then, if necessary, it computes the pages accessed through the indirection array interaction\_list. Finally, it requests the updates to each page of the coordinate array that will be accessed by the executing thread. To improve performance, Validate aggregates requests for multiple pages from the same thread.

### 4.1.2 Augmented Run-Time System

The TreadMarks run-time system was augmented in order to take advantage of the access information provided by the compiler. We concentrate here on the support for communication aggregation for irregular accesses. Support for regular accesses and other optimizations was described in earlier work [DCZ96]. Figure 4.3 provides a summary of the Validate interface for both regular and irregular accesses.

To support aggregated communication, Validate can fetch multiple data objects at the same time. Thus, it takes a variable number of arguments. The first argument
/* fetch_pages is the list of pages to be fetched  
   pages[sch] is the list of pages associated with each schedule */

Validate(va_list) /* Handles variable number of arguments */
{
    va_list *desc_ptr
    int number /* number of descriptors */
    int descriptor

    va_start(desc_ptr)
    fetch_pages = NULL
    number = va_arg(desc_ptr, int)

    for (descriptor = 1; descriptor <= number; descriptor++)
    {
        int type = va_arg(desc_ptr, int) /* descriptor type - DIRECT or INDIRECT */
        char *base = va_arg(desc_ptr, char *) /* base address of shared data */
        RSD section = va_arg(desc_ptr, RSD) /* section of shared data or indirection array */
        int access_type = va_arg(desc_ptr, int) /* READ, WRITE, READWRITE, WRITE_ALL, or READWRITE_ALL */
        int sch = va_arg(desc_ptr, int) /* schedule number */

        if (type == INDIRECT)
        {
            if (modified(section))
            {
                pages[sch] = Read_indices(base, section)
                Write_protect(section)
            }
        }
        else
        {
            pages[sch] = pages in section
            fetch_pages += pages[sch] that are invalid
        }
    }

    Fetch_diffs(fetch_pages)
    Apply_diffs(fetch_pages)

    for (descriptor = 1; descriptor <= number; descriptor++)
    {
        if (access_type == WRITE || access_type == READWRITE)
            Create_twins(pages[sch])
    }

    va_end(desc_ptr)
}

Figure 4.3 : Augmented Run-Time Interface for Indirect Accesses
is the number of access descriptors that follow. There is an access descriptor for each
data object. An access descriptor consists of type, base, section, access_type, and
schedule_number. The type specifies the descriptor type. It is either DIRECT for
regular accesses or INDIRECT for accesses through an indirection array. The base is
the address of the shared data structure being accessed. For an irregular problem,
the section is the section of the indirection array used to access the shared data, or
the section of shared data itself in the case of regular accesses. The access_type is
one of READ, WRITE, or READ&WRITE. Direct accesses have two additional access types,
WRITE_ALL and READ&WRITE_ALL, which indicate when every element in the section is
known to be written at compile-time. The run-time system can use this information
to reduce consistency maintenance overhead by eliminating twinning on those pages
that are completely written. The schedule_number is an identifier for the set of pages
to be fetched.

If the descriptor type is INDIRECT and the section of the indirection array has
been modified since the last call to Validate, the modified function returns true, and
pages[sch] is recomputed. Both local and remote modifications cause the modified
function to return true. The Read_indices procedure recomputes the list of pages,
pages[sch], using base and section. After pages[sch] has been computed, the
pages in section are write protected. A more sophisticated version of this approach
could use differencing (comparing an old version of the pages containing the indirection
array to the current one) to incrementally recompute the page sets, but our current
implementation does not do so. Those pages in pages[sch] that are invalid are added
to fetch_pages, the list of pages to be fetched.

Fetch_diffs requests the diffs required to update the pages in fetch_pages.
All of the diff requests to the same thread are aggregated into a single message.
Apply_diffs waits for the diffs to arrive, and applies them to the appropriate page.

After updating the pages of shared data, consistency actions are performed pre-
emptively in order to avoid write detection overhead during execution. The Tread-
Marks multiple-writer protocol requires an unmodified copy (a twin) of the page to be maintained for every page that is modified, unless the page is guaranteed to be modified in its entirety. Validate performs Create_twins on pages[sch] if the corresponding descriptor has an access_type of WRITE or READ&WRITE. Create_twins makes a twin of each page in pages[sch], and enables write access to these pages. This avoids the memory protection violation to create the twin.

4.1.3 Compiler Analysis

The purpose of our compiler analysis is to provide access pattern information to the run-time system. This involves both determining what data is accessed, and determining at which point to supply the access information to the run-time system.

The access analysis simply determines the indirect array used to access shared data, and the part of the indirect array being accessed. This is usually a regular section [CK88], and hence can be handled by regular section analysis [HK91]. Regular section descriptors (RSDs) are used to concisely characterize the array accesses in a loop nest. RSDs represent the accessed data as linear expressions of the upper and lower loop bounds along each dimension, and include stride information. Our approach also naturally extends to multiple levels of the indirect in the access pattern without additional mechanisms. In contrast, the inspector-executor approach requires several inspector loops to be generated for such access patterns [DHSK95].

Our compiler feeds the access information by calling Validate at conveniently defined fetch points, and relies on the run-time system to determine when to create the schedule. Because the run-time system already implements a shared memory interface, it can detect both the local modifications to the index array and those from the remote threads. In contrast, the inspector-executor approach requires sophisticated compiler analysis to pull the inspector as far forward as possible in the program. The fetch points include all synchronization points in the program, plus procedure entries and conditional statements. Synchronization points are natural fetch points because
in shared memory programs, these are the points where shared memory states are made consistent across the threads, so that each thread can determine what data to fetch in order to make local memory consistent. The synchronization points in OpenMP include the start and end of parallel regions in addition to barriers and locks. Conditional statements or procedure calls limit the region of code for which our simple compiler can summarize access information, so that they are also included as fetch points. We therefore analyze code segments between consecutive fetch points, and provide the run-time system with a description of the accesses at the beginning of the segment.

The compiler analysis and the transformation can be summarized as follows. We concentrate here on the additions to the analysis necessary for handling indirect accesses (see [DCZ96] for details on how regular accesses are handled). Let \( V \) be the set of shared variables, and let \( F \) be the set of fetch points. Access analysis generates a summary of shared data accesses associated with each element of \( F \), and the type of such accesses. For each statement \( p \) in the program, for each definition or reference in \( p \) to an indirection array, a section is constructed. A \{READ\}, \{WRITE\}, or \{READ\&WRITE\} tag is associated with the section depending on the access type. This section is associated with each element of \( F \) that directly precedes \( p \).

During the program transformation phase, for each \( f \) in \( F \), if there are access descriptors associated with \( f \), a \texttt{Validate} is inserted at \( f \). Each access descriptor is then supplied as a parameter to \texttt{Validate} with either a \texttt{DIRECT} or \texttt{INDIRECT} type. If the type is \texttt{INDIRECT}, the base address of the shared data is supplied to the \texttt{Validate} call, along with the RSD for the indirection array as the \texttt{section} parameter.

See Figure 4.2 for the results of the analysis and the transformations on the Moldyn program. The relevant fetch point is the entry to the procedure \texttt{ComputeForces}. The sections of the indirection array \texttt{interaction_list} are used to fetch the corresponding page sets of the data array \texttt{x}. After the initial execution of \texttt{Validate}, \texttt{interaction_list} is write protected. When the interaction list is modified after a
number of iterations specified by UPDATE\_INTERRUPT, a memory protection violation occurs. The handler for this memory protection violation sets a flag. During the next execution of Validate, if the flag is set, modified clears the flag and returns true, and Validate recomputes the set of pages that must be fetched.

4.2 CHAOS

We evaluate our approach by comparing it with the inspector-executor approach. The comparison is between our automatically optimized shared memory programs and the hand-tuned inspector-executor versions of the same applications. The CHAOS run-time library [SPM+94] is used to support the inspector-executor model. There are three steps in solving irregular problems in CHAOS, namely, data and iteration partitioning, the inspector, and the executor.

CHAOS supports a number of parallel partitioners that partition data arrays using heuristics based on spatial position, computational load, etc. The partitioner returns a translation table, which contains an irregular assignment of array elements to threads. A translation table lists the home thread and offset address of each data array element. Depending on storage requirements, the translation table can be replicated, distributed regularly, or stored in a paged fashion. This table is used by the inspector to create the communication schedule. If the translation table is not replicated, communication may be necessary in the inspector. The loop iterations are partitioned by the almost-owner-computes rule, which assigns an iteration to the thread that owns a majority of data array elements accessed in that iteration. The data array can be remapped, so that data elements owned by a thread are adjacent in memory. Remapping has the potential advantage that the memory requirement for a thread is proportional to the size of the data partitions assigned to it.

The Recursive Coordinate Bisection (RCB) partitioner is one specialized partitioner supported by the CHAOS library that partitions nodes according to their physical positions. When simulating physical systems, particles close to each other in
the physical space are more likely to interact with each other, or to be connected with each other. RCB results in less communication than a simple BLOCK or CYCLIC partition on these applications.

Each thread executes the inspector to construct its communication schedule. A communication schedule specifies which data is communicated and which threads are involved. The inspector constructs the communication schedule by first determining the data read and written on each thread and then consulting the translation table to determine the global placement of this data according to the partition. An important optimization in the inspector is to eliminate duplication. Duplication occurs when a data array element is pointed to by many elements in the indirection array. Removing duplication can dramatically reduce the amount of data communicated. A hash table whose size is proportional to the size of the data array is employed to eliminate duplicates. Because of the time to hash the indirection array, and the time to look up the translation table, the inspector can be expensive. However, this overhead can be amortized if the indirection array remains unchanged for a long period of time.

The executor uses the communication schedule generated by the inspector to gather and scatter data. Gather fetches off-thread data, and scatter propagates modifications to off-thread data back to their owners.

4.3 Experimental Evaluation

We use an 8-processor IBM SP2 running AIX version 3.2.5. Each processor is a thin node with 64Kbytes of data cache and 128Mbytes of main memory. Interprocessor communication is accomplished over the IBM SP2 high-performance switch. Unless indicated otherwise, all results are for 8-processor runs.

We compare the compiler-optimized OpenMP programs with the hand coded CHAOS programs, as well as the base OpenMP programs. The compiler-optimized OpenMP programs include optimizations for both regular and irregular access patterns. Tables 4.1 and 4.2 present the execution times, speedups, number of messages
and the amount of data communicated at 8 processors for the two applications discussed in this chapter.

<table>
<thead>
<tr>
<th>Update frequency</th>
<th>Time (sec)</th>
<th>Speedup</th>
<th>Messages</th>
<th>Data (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Every 20 iterations (seq = 267.2 sec)</td>
<td>CHAOS</td>
<td>44.9</td>
<td>6.0</td>
<td>15704</td>
</tr>
<tr>
<td></td>
<td>Tmk base</td>
<td>42.3</td>
<td>6.3</td>
<td>62149</td>
</tr>
<tr>
<td></td>
<td>Tmk optimized</td>
<td>37.7</td>
<td>7.1</td>
<td>14528</td>
</tr>
<tr>
<td>Every 15 iterations (seq = 365.8 sec)</td>
<td>CHAOS</td>
<td>61.7</td>
<td>5.9</td>
<td>16255</td>
</tr>
<tr>
<td></td>
<td>Tmk base</td>
<td>56.4</td>
<td>6.5</td>
<td>70230</td>
</tr>
<tr>
<td></td>
<td>Tmk optimized</td>
<td>48.9</td>
<td>7.5</td>
<td>14687</td>
</tr>
<tr>
<td>Every 11 iterations (seq = 467.3 sec)</td>
<td>CHAOS</td>
<td>78.2</td>
<td>6.0</td>
<td>16806</td>
</tr>
<tr>
<td></td>
<td>Tmk base</td>
<td>68.1</td>
<td>6.9</td>
<td>71788</td>
</tr>
<tr>
<td></td>
<td>Tmk optimized</td>
<td>60.4</td>
<td>7.7</td>
<td>14871</td>
</tr>
</tbody>
</table>

Table 4.1: Moldyn - 8 processor results. The interaction list is updated at varying intervals.

4.3.1 Moldyn

Moldyn is a molecular dynamics simulation. Figure 4.1 illustrates the program structure of Moldyn, and the force computation subroutine. Section 3.8 discusses Moldyn’s background and algorithm in detail. Basically, the program maintains an interaction list of all the pairs within the cutoff distance, and iterates over this list at each timestep. The interaction list is used as an indirection array to identify interacting partners. Since molecules change their spatial location every iteration, the interaction list must be periodically updated.

The CHAOS program uses the RCB partitioner to assign molecules to threads. This partition lasts through the execution. When the interaction list is updated, the program must again call the inspector to identify interacting partners. This call is inserted in the main program, right after the call to subroutine build_interaction_list.
In ComputeForces, each thread uses the schedule created by the inspector to gather remote values of $x$ and $forces$ before the main loop. Both $x$ and $forces$ are modified elsewhere, necessitating the gather. After the main loop, the threads again use the schedule to scatter values of $forces$ that will be read by other threads.

The OpenMP program also uses the RCB partitioner. The coordinate array $x$ and the $forces$ array are shared. The $Validate$ on $x$ that was inserted by the compiler appears at the beginning of the subroutine $ComputeForces$, before the loop over the interaction list. Changes to the interaction list are detected by write protecting the pages it occupies inside $Validate$. An explicit inspector call is hence not needed. The accumulation of the forces on each molecule is an OpenMP reduction operation. OpenMP implements reduction by first accumulating each thread’s contributions in a local array, then updating the shared array in a pipelined fashion.

In OpenMP, the local force array used in reduction is indexed by the molecule number without any translation. Thus the local force array is proportional in size to the total number of molecules. In CHAOS, remapping creates an analog to the local force array that is proportional in size to the molecules assigned to that thread plus the molecules that it interacts with. For the default data set, which we used in our experiments, between 31% and 53% of the molecules interact. Consequently, remapping has little effect on the memory utilization of the CHAOS program.

Results

We simulated 16384 particles for 40 iterations, varying the number the times the interaction list is updated from 1 through 3. The results are presented in Table 4.1. The data initialization and the data partitioning for the parallel programs are not timed for either the sequential or parallel versions.

We first present results for the case where the interaction list is updated once, at the 20th iteration. The sequential program without any calls to CHAOS or any OpenMP annotations runs for 267 seconds. The OpenMP execution time on a single
processor is almost identical to that of the sequential program, spending only 0.4 seconds to check the indirection lists. On the other hand, the CHAOS program runs longer on a single processor than the sequential program, because it spends 6.2 seconds in the inspector.

At eight processors, the CHAOS program runs for 44.9 seconds. We were unable to use a replicated translation table, owing to the amount of memory that it required. The translation table is hence distributed, necessitating communication in the calls to the inspector. In the case where the interaction list is updated at the 20th iteration, the inspector is called twice, including once at the beginning of the program. Each thread spends 4.6 seconds in the inspector. Exchanging the translation tables causes the transfer of 85Mbytes of data in 878 messages.

The base OpenMP program (without any compiler support) runs for 42.3 seconds on eight processors. OpenMP is able to achieve a performance comparable to CHAOS because of the large problem size, and the good data locality provided by the RCB partitioner. However, the number of messages sent in OpenMP is three times more than that in CHAOS. The reason is that the underlying TreadMarks run-time obtains data one page at a time, while CHAOS sends all the data needed by a thread in a single message.

With the compiler optimizations, the OpenMP running time comes down to 37.7 seconds, which is an 11% improvement over the base OpenMP. Of this improvement, 7 percentage points come from the communication aggregation for regular accesses. The remaining 4 percentage points come from the compiler inserted call to validate for the indirect accesses. The optimized OpenMP program spends 0.6 seconds in validate to check the indirection array. The optimized OpenMP program sends 23 Mbytes less data than the base OpenMP program because the compiler recognizes the reduction on the force array. During the pipelined update of the shared force array, each thread reads and writes the entire section of the array. As a result, the compiler can flag via validate that the entire page, and not the diff, must be sent
on a diff request.

When the interaction list is updated more often, the running times increase because of the time taken to rebuild the interaction list. CHAOS suffers from having to rerun the inspector. When the interaction list is updated every 11 iterations, CHAOS spends 9.2 seconds per thread on average in the inspector, while OpenMP spends only 0.8 seconds in scanning the indirection list. As a result, the optimized OpenMP program is 23% faster than CHAOS.

4.3.2 NBF

NBF is the kernel of a molecular dynamics simulation. As explained in Section 3.9, NBF keeps a list of interacting partners for each molecule, instead of keeping a list of pairs of interacting molecules like Moldyn. In our experiments, the partner list is static. Each molecule has approximately the same number of partners, and the partners of each molecule spread evenly in about 2/3 of the total space.

In the CHAOS program, the inspector is called at the beginning of the program, outside the loop simulating the time steps. At the start of each time step, a gather is called to collect the updated values of coordinates from remote threads. A scatter is invoked at the end of each time step to propagate the modifications to the force array.

A Validate is performed at the start of each time step to fetch the updated values of the coordinate array in the OpenMP program. Like Moldyn, the update to the forces is an OpenMP reduction operation, and is implemented first accumulating each thread's contributions in a local array, then updating the shared array in a pipelined fashion.

For the data set which we used in our experiments, 84% of the molecules interact. Consequently, remapping yields little reduction in the memory utilization of the CHAOS program.
Results

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Time (sec)</th>
<th>Speedup</th>
<th>Messages</th>
<th>Data (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 × 1024</td>
<td>CHAOS</td>
<td>10.9</td>
<td>7.2</td>
<td>2014</td>
</tr>
<tr>
<td></td>
<td>Tmk base</td>
<td>19.6</td>
<td>4.0</td>
<td>34421</td>
</tr>
<tr>
<td></td>
<td>Tmk optimized</td>
<td>12.1</td>
<td>6.5</td>
<td>4817</td>
</tr>
<tr>
<td>(seq = 78.3 sec)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64 × 1000</td>
<td>CHAOS</td>
<td>10.6</td>
<td>7.2</td>
<td>2014</td>
</tr>
<tr>
<td></td>
<td>Tmk base</td>
<td>19.4</td>
<td>3.9</td>
<td>36278</td>
</tr>
<tr>
<td></td>
<td>Tmk optimized</td>
<td>12.3</td>
<td>6.2</td>
<td>4920</td>
</tr>
<tr>
<td>(seq = 76.5 sec)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32 × 1024</td>
<td>CHAOS</td>
<td>5.5</td>
<td>7.1</td>
<td>2014</td>
</tr>
<tr>
<td></td>
<td>Tmk base</td>
<td>9.1</td>
<td>4.3</td>
<td>18095</td>
</tr>
<tr>
<td></td>
<td>Tmk optimized</td>
<td>6.2</td>
<td>6.3</td>
<td>3851</td>
</tr>
<tr>
<td>(seq = 39.1 sec)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: NBF Kernel - 8 processor results.

We ran NBF with varying numbers of molecules for the input problem size (see Table 4.2). Each molecule is represented by a double precision floating point number. Each molecule has 100 partners evenly distributed in the memory. The test runs for 11 iterations, of which the last 10 iterations are timed. Thus, the results include neither the time to perform the inspector in the CHAOS version nor the time for checking the partner array in the OpenMP program.

The unmodified (original) sequential program runs for 78.3 seconds with a problem size of 64×1024. The single-thread OpenMP execution time is almost identical to that of the sequential program, spending only 0.001 seconds in scanning the indirection array. On the other hand, the CHAOS program runs longer on a single processor than the sequential program, because it spends 7.3 seconds in the inspector.

At eight processors, the CHAOS program and the optimized OpenMP program run for 10.9 seconds and 12.1 seconds, respectively. The inspector is not included in the timing for CHAOS. The main reason for the 10% difference is that CHAOS
pushes the data to the threads that will use it in one message, while OpenMP uses request–response communication (necessitating two messages). The 13% extra data sent in OpenMP is due to false sharing.

Although we excluded the time to run the inspector from the timing, it is important to note that at eight processors, the CHAOS program spends 5.2 seconds per thread to create the schedule. In contrast, the OpenMP program only spends 0.3 seconds going through the indirection array.

The compiler optimizations reduce the execution time of the base OpenMP version by 38%. Of this reduction, 34 percentage points come from optimizations in the regular part of the code, such as the pipelined reduction. These optimizations reduce both the number of messages and the amount of data sent in the program. The remaining 4 percentage points come from prefetching the data for the irregular accesses at the beginning of each time step.

Reducing the problem size to \(32 \times 1024\) does not affect the relative performance of OpenMP and CHAOS much. The difference in performance comes from OpenMP having to request data, as in the case of the \(64 \times 1024\) problem size. Changing the data set size to \(64 \times 1000\), we introduce false sharing at the boundary between pairs of threads. In this case, the optimized OpenMP program is 14% slower than the CHAOS program, because of the extra messages and data caused by false sharing. However, the cost of the inspector in CHAOS overshadows the performance loss from false sharing in OpenMP.

## 4.4 Related Work

A large number of studies have been published on the performance of distributed shared memory and inspector-executor systems, but, to the best of our knowledge, only one paper has been published comparing the two approaches. Mukherjee et al. [MSH+95] compare the CHAOS inspector-executor system to the TSM (transparent shared memory) and the XSM (extendible shared memory) systems, both imple-
mented on the Tempest interface [RLW94]. Three applications are used: Moldyn, unstructured, and DSMC, and the comparison is done on a 32-processor CM-5. They conclude that TSM is not competitive with CHAOS, while XSM achieves performance comparable to CHAOS after introducing several special-purpose protocols.

Our study differs from the cited paper in several aspects. First, our transparent shared memory system (TreadMarks) performs significantly better than TSM. We attribute this difference in performance to TreadMarks' use of lazy release consistency and multiple writer protocols, in contrast to the sequential consistency and single writer protocols used in TSM. Second, we use a compiler to optimize the shared memory programs, rather than relying on hand-coded special-purpose protocols. As indicated in our study, the compiler analysis necessary is relatively straightforward.

Our study is also related to the many papers on prefetching and aggregation. In particular, Mowry et al. [MDK96] use a somewhat similar strategy to prefetch and aggregate disk requests for sequential programs, and Dwarkadas et al. [DCZ96] study prefetching and aggregation for regular applications in software distributed shared memory systems.

4.5 Summary

We have described an integrated compile-time/run-time optimization for executing irregular OpenMP programs on networks of workstations. This approach is based on a modified software distributed shared memory layer, and fairly simple compile-time support. The only required compile-time support is regular section analysis of the indirection arrays. Run-time support for dynamic detection of changes to the indirection array, as well as to the shared data, eliminates any unnecessary computation and communication. Furthermore, the communication by each thread is aggregated into fewer message exchanges.

We measured this approach for two irregular applications, Moldyn and NBF, on an 8-processor IBM SP2. We find that the OpenMP based approach has similar
performance to the inspector-executor method supported by the CHAOS run-time library, while requiring much simpler compile-time support. For Moldyn, it is up to 23% faster than CHAOS, depending on the input problem’s characteristics; and for NBF, it is no worse than 14% slower. The advantage of the OpenMP based approach increases as the frequency of changes to the indirection array increases. The disadvantage of this approach is the potential for false sharing overhead when the data set is small or has poor spatial locality. In addition, in both Moldyn and NBF, the OpenMP based approach eliminated substantial inspector overhead. For both applications, the OpenMP based approach is always faster than CHAOS if we include the execution time of the inspector.
Chapter 5

Improving Scalability

This chapter studies methods to scale OpenMP to a network of large numbers of processing nodes. Software distributed shared memory (SDSM) systems provide a shared memory interface on top of distributed memory machines. Beginning with Li and Hudak [LH89] in 1985, designers of software distributed shared memory (SDSM) protocols have focused much of their efforts on reducing the number of messages to maintain memory consistency. We proposed an algorithm in Chapter 4 that aggregates messages in SDSM systems, and we demonstrated significant improvement in execution time on a network of eight computers. Indeed, protocol comparisons on small networks with eight to sixteen nodes have consistently shown a strong correlation between the execution time of a program and the number of messages exchanged by the protocol. However, our recent study [DLHL+00] shows that on large networks, the distribution of the number of messages handled by each node is no less important than the total number of messages.

This result should not be surprising because load balancing is crucial for parallel applications to achieve good speedup. Application programmers take great care to make sure that the computation is distributed evenly among the threads. For the same reason, the amount of communication should also be balanced between different threads. On a small scale SDSM system, because the total amount of communication is limited, the magnitude of communication load imbalance is often not big enough to adversely affect the execution time of the program. However, in a large scale SDSM system, the lack of load balance may cause a node to be overwhelmed by a huge amount of simultaneous request messages, resulting in severe contention on
that node. Contention intensifies with the increase of processing nodes, and causes prolonged memory latency.

In the same study [dLHL+00], we also find that the sequential sections in parallel programs are major sources of contention. Parallel programs usually have sequential sections for initializing data, and for parts of the code that are either too complicated or too expensive to run in parallel. In the shared memory paradigm, the sequential sections are executed by one thread, and that is the master thread in OpenMP. The data written during the sequential section are accessed by all threads in the parallel computation that follows. Each thread may access a different part of the data, but they all go to the last writer which is the master thread for the updates.

We propose a method that completely eliminates contention caused by the sequential sections of parallel programs. Our solution replicates the sequential execution on all threads, allowing each thread to replicate any modifications on a local copy of the shared data. This completely eliminates the communication to propagate changes made during the sequential execution, and eliminates contention at its source. During the replicated sequential execution, we take advantage of the fact that every thread executes the same code: When several threads need the same data, only one request is sent, and the result is multicast to all threads participating in the parallel computation. Because a multicast message has the same cost as a point-to-point message, this lowers the overhead of replicated sequential execution to a level similar to the non-replicated version. Finally, we use the fork-join programming model of OpenMP to automatically identify the sequential sections at runtime, so that our optimization is completely transparent to the OpenMP application programmer. Our optimization does not rely on compile-time data analysis, and is completely integrated into the SDSM runtime. As a result, even irregular pointer-based applications can benefit from our optimizations.

The rest of this chapter is organized as follows. Section 5.1 explains the cause of contention, and its effect on SDSM systems. Section 5.2 discusses our design choices.
Section 5.3 details the implementation of our system. The results are presented in Section 5.4, and related work is presented in Section 5.5. We conclude in Section 5.6.

5.1 Contention

This section first illustrates the effect of contention on SDSM systems, then explains why sequential sections are major sources of contention.

We define contention as simultaneous requests on a node. On our platform, contention can be attributed to limitations in the node or the network. In the former case, the time that the node requires to process a request is longer than it takes for the next request to arrive. In the latter case, the node fails to push out responses fast enough due to bandwidth limitations in the network link. Most systems, under this condition, wait for the interface to free an entry in its output queue.

Although contention happens at the node holding the updates, its effects are felt by the requester. As most SDSM systems handle requests in the same order in which they arrive, the service time for a request that arrives at a node with \( n \) outstanding request is increased by the time required to process all preceding requests.

![Figure 5.1: Contention.](image-url)

Figure 5.1a illustrates the case where no contention takes place. The response
time, namely the time between the request is sent and the time by which the reply is received, consists of the service time and the wire time. In Figure 5.1.b on the contrary, contention occurs when both p1 and p2 request from p0 simultaneously. While p1 gets the reply back in roughly the same time as before, p2's request is not processed until p1 is served.

A major cause of contention are the sequential sections in parallel programs. Sequential sections are inevitable in parallel programs. Initialization and irregular load balancing codes are usually run sequentially. These sections usually take a very small portion of the execution time, and parallelizing them could be both complicated and unprofitable. A parallel region whose amount of work varies from iteration to iteration may also be executed sequentially when the amount of work associated with it is below a threshold. OpenMP provides the if clause on the parallel construct to parallelize code conditionally at runtime.

In the shared memory paradigm, the sequential sections are executed by one thread which is the master thread in OpenMP. The result of the sequential computation automatically becomes visible to the other threads via proper synchronizations. The data written during the sequential section are often accessed by all threads in the parallel computation that follows. Each thread may access a different part of the data, but they all go to the last writer for the updates.

Consider for example, Barnes-Hut, a N-body simulation program from the SPLASH-2 benchmarks [WOT+95]. A shared oct-tree is used in Barnes-Hut for load balancing purpose. The tree is rebuilt at the beginning of each iteration by reading all the particles updated in the previous iteration. Because the tree construction takes 0.3\% of the sequential execution time and is very complicated to parallelize, it is executed sequentially. At the beginning of the force evaluation phase that follows tree building, each thread performs a top-down traversal of the tree to find its own particles. Contention occurs here because every thread simultaneously requests the new version of the tree from the master thread.
Contention may also occur between two consecutive parallel sections, but it is not inherent in this situation. Because parallel regions distribute the computation and data accesses among different threads, there need not be a single thread to which everyone request for data simultaneously. Careful programming practice should be able to avoid contention in this case.

5.2 Design

Our approach addresses the contention caused by sequential sections in parallel programs. This section justifies our design choices. The detailed implementation is presented in Section 5.3.

Contention is caused by too many requests arriving at a thread simultaneously. Because the data modified during the sequential section are likely to be accessed by multiple threads in the parallel section, an obvious solution would be to predict which threads are going to access which pages, and multicast the pages to those threads accordingly. With IP multicast, a single send causes a message to be delivered to multiple threads, and the cost of sending a multicast message is identical to the cost of sending a point-to-point message. If implemented efficiently, this approach could dramatically reduce contention. However, there are two issues that limit the applicability of this method.

First, this method requires great effort to predict which threads are accessing the data. Compile-time analysis can predict data access patterns [LC91, AL93b, AL93a, AKN95], but it is limited to regular access patterns, which do not include pointer based applications such as Barnes-Hut. Runtime speculations have also been proposed to predict accesses based on previous history [SB98], but such an approach works only for those applications that have repetitive access patterns from one iteration to another.

Second, the underlying network protocol does not support efficient implementation of this method. Instead of going to a certain node, multicast messages are sent to
a multicast group, and the underlying network and the network interface guarantees that each process in the multicast group receives the message. A thread is required to join a multicast group in order to receive messages sent to the group, and it may also explicitly leave the group. In a switched network, changing membership of the multicast group requires a message to be sent to the switch, alerting the switch of the change. Most implementations of IP multicast allow each process to join a maximum of 20 multicast groups simultaneously. In Barnes-Hut, with 128K particles on 32 nodes, there are thousands of different access patterns, half of which each thread has to join, making changing the multicast groups on the run prohibitively expensive. In general, the number of different access patterns may grow exponentially with the number of nodes, so that multicasting to each different access group is impractical on large scale systems.

Another possible solution is to broadcast all data modified during the sequential execution to everyone participating in the parallel computing. However, this method disrupts the threads that do not access the data, and does not scale to large systems. In Barnes-Hut, except for the few pages containing the top level nodes of the tree that are accessed by all nodes, most of the tree pages are accessed by roughly half of the nodes. With a larger problem size and more processing nodes, we expect most pages to be accessed by an ever smaller portion of the nodes. As a result, a simple broadcast wastes too much bandwidth and processing power on the receivers of the message.

Compared to the aforementioned approaches, our approach completely eliminates contention at its source. It does not rely on compiler analysis, and makes efficient use of multicast. Our solution replicates the sequential execution on all threads, allowing each thread to replicate the modification on the local copy of the shared data. This completely eliminates the communication to propagate changes made during the sequential execution, and eliminates contention at its source.

During the replicated sequential execution, we take advantage of the fact that
every thread executes the same code and accesses the same data, so that when several threads need the same data, only one request is sent as a result, and the reply is multicast to all threads participating in the parallel computation. Because a multicast message has the same cost as a point-to-point message, this lowers the overhead of replicated sequential execution to that of the non-replicated version. Multicast is crucial to the efficiency of the replicated execution. Without multicast, the amount of communication would multiply. Furthermore, since every thread executes the same code, and tend to request for the same data simultaneously, contention is very likely to happen.

One concern is that turning each unicast reply into a multicast message would consume too much network bandwidth and result in performance penalty by saturating the network. We can avoid the congestion by allowing one multicast message at a time. The threads simultaneously wait for the updates for one page, and no new requests can be generated before the current page is brought up-to-date. Because of the replicated execution, no concurrency is lost by limiting the number of outstanding multicast messages.

For Barnes-Hut, our method would let every thread execute the sequential tree building. Because a thread reads all the particles to build the tree, the pages containing the particles are broadcast during the replicated execution. As a result, by the end of the tree building phase, everyone has a complete new tree cached locally, and the up-to-date values of the particles.

5.3 Implementation

In addition to efficiency, we also want our replicated execution to be completely transparent to the application programmer. There are three important problems to be solved in order to achieve these goals.

First, the sequential sections of the programs must be identified and replicated without programmer intervention. Second, the replicated execution should integrate
seamlessly with the shared memory consistency model. Finally, to efficiently implement multicast during the replicated execution, we make sure when multiple threads miss on the same data, only one request is sent and the reply is multicast to all threads.

Our implementation is based on the OpenMP API. The optimizations are purely runtime, without any change to the OpenMP translator. The underlying TreadMarks runtime library is modified to support the replicated sequential execution and the multicast.

5.3.1 Replicate Sequential Sections

OpenMP requires the parallel regions in a program to be explicitly marked, and then the compiler generates fork-join parallel programs that issue a fork before the parallel region, and a join after the parallel region. Our approach automatically replicates the execution between a join and the next fork at runtime. It does not require any extra effort from either the OpenMP programmer or the compiler to identify sequential sections. Shared memory allocation, input and output instructions are not duplicated. We use the runtime variables provided by OpenMP to guard each memory allocation, input or output instruction so that it is executed only by the master thread during the sequential execution.

At the join, in a system without replicated sequential execution, the master thread waits until all threads have finished the previous parallel section, and then goes on to execute the sequential code. A slave thread sends a message to the master at the join, then goes to sleep waiting for the next fork. In the replicated implementation, the join is changed from a one way communication to a barrier. The master thread waits until all threads have arrived at the join, then issues a departure message so that everyone continues to the replicated sequential section. After the join, all threads are guaranteed to see all modifications made before the join.

At the fork, threads wait until all other threads have finished the sequential exe-
cution to proceed to the next parallel section. No memory coherence information is exchanged at the fork, because every thread executes exactly the same code during the replicated sequential execution. As in a normal fork, the slaves obtain the parameters from the master thread. Although the sequential execution is replicated, it is still necessary to pass pointers to the shared data from the master to the slaves at the fork because the shared data is allocated by the master thread.

Since fork and join mark the boundaries of the sequential execution, operations related to other aspects of the implementation are also performed at these points.

### 5.3.2 Integrating with the Consistency Protocol

Replicating the sequential execution may result in inconsistent shared memory states, because there is no synchronization within the sequential code. A race condition occurs especially when there are unsynchronized accesses to the shared data, of which at least one is a write. A race condition resulting from replicated execution is illustrated in Figure 5.2. The shared variable \( x \) has a initial value of one, and is increased by one during the sequential section. If the sequential section is executed by both p0 and p1, the final value of \( x \) may be either two or three. The value could be three if the already increased value of \( x \) propagates from one process to another before the latter reads \( x \) for increment.

![Figure 5.2: Race condition.](image)

In general, the race condition really results from the shared memory mechanism that automatically propagates the modifications to shared variables, which should be
disabled during the replicated sequential sections. An implementation that disables the consistency mechanism during the replicated execution must integrate seamlessly with the DSM protocol, both for correctness and for efficiency.

In TreadMarks, we avoid race conditions by taking advantage of the release consistency (RC) protocol employed by TreadMarks. With release consistency, modifications to the shared memory do not have to be visible to others until the next synchronization point. This allows us to delay propagating the modifications made during the sequential section until the subsequent fork, thus eliminating race conditions. However, a specific implementation may propagate the update anytime before that, thus still causing race conditions. In TreadMarks, lazy diff creation, an optimization that aggregates modifications made during different intervals in one diff, may cause such early updates. A simple solution to this problem is to disable lazy diff creation, and create diffs for all modifications before entering the sequential section, but experiments [KDCZ94] have shown that lazy diff creation is important for programs with little false sharing, and should not be disabled. Our implementation avoids race conditions without disabling this optimization.

The updates in TreadMarks are encoded in diffs. A diff is a runlength record of the modifications to the page. In order to capture the modifications to a shared page, a page is initially write-protected. At the first write, a protection violation occurs. The SDSM software makes a copy of the page (a twin), and removes the write protection so that further writes to the page can occur without any SDSM intervention. The twin and the current copy can later be compared to create a diff. Each thread allocates a certain amount of memory for diffs. Once the diffs space is filled, all threads participate in a global garbage collection to clean up the diffs and the associated consistency information.

The lazy implementation of RC allows diff creation to be postponed until the modifications are requested. In TreadMarks, a diff is created when a request asks for it, or when a write notice from another thread invalidates the page. In the latter
case, it is essential to make a diff in order to distinguish the modifications made by
the different threads. Lazy diff creation results in a decrease in the number of diffs
and an attendant improvement in performance [KDCZ94].

To avoid race conditions during replicated sequential execution in the presence of
lazy differencing, all dirty pages are write protected before entering the sequential section.
During the sequential execution, the first write to the page causes a page fault upon
which a diff is created for the page. A twin is allocated after the diff creation and
the page is made writable. A diff request during the sequential section will obtain
this diff, which contains only the modification made before the sequential section. At
the end of the replicated execution, the remaining write protected dirty pages are
unprotected and returned to their normal state.

5.3.3 Multicast Implementation

Our implementation is based on IP multicast. There is one multicast group during the
execution which every thread joins at the beginning of the program. The main task of
the multicast implementation is to efficiently add reliability to IP multicast. How to
provide reliable and efficient multicast at either the transport layer or the application
layer is an ongoing research topic. However, this paper does not focus on multicast
protocols - we just build a simple multicast mechanism tailored to our system that
provides congestion avoidance and flow control. In the future, this multicast protocol
could be improved, or suitable protocols developed somewhere else could be used in
our system.

Because a multicast message is sent to everyone in the group, in order to avoid
congestion, we guarantee that only one thread sends at a time. When multiple threads
fault on a page during the replicated execution, a single thread sends a diff request
on behalf of all faulting threads. Diff requests from different threads are serialized at
the master thread. A request is first sent via a point-to-point message to the master,
and the master thread multicasts the request to all threads, making sure only one
page is being served at a time. After receiving the multicast diff request, the threads multicast the missing diffs in turn, with one thread sending at a time. Flow control is achieved by requiring each thread to multicast an acknowledgment after receiving all the diffs sent by the preceding thread in line. When a thread has diffs to send, the diff reply message serves as acknowledgment, otherwise a null acknowledgment message is sent.

Another important problem is to efficiently choose the thread that sends the diff request during the replicated sequential execution. Because threads access different data during the parallel sections, a page can be valid on certain threads and invalid on the others. When multiple threads fault on one page, our implementation chooses the one with the lowest thread identifier to send the request on behalf of all faulting threads.

In order to choose the sender without additional communication, a thread must know whether a page is valid or not on other threads, as well as which diffs are missing on other threads. Since this information is not available in the original TreadMarks system, we augment each page with an array of valid notices, with one entry corresponding to each thread. A valid notice records the timestamp of the latest interval during which the page is brought up-to-date by a thread. By comparing the write notices and the valid notices on a page, a thread infers which threads fault on this page, and which diffs are missing on those threads. Like write notices, valid notices are exchanged at synchronizations. But because we only use valid notices during the replicated execution, they are exchanged only at the join before a sequential section.

This multicast implementation has some potential overhead – first, the additional forwarding of requests to the master thread and the null acknowledgment messages; second, the overhead for choosing the sender of the multicast request; third, the overhead for exchanging the valid notices at the entry of sequential sections. In addition, some concurrency is lost compared with diff requests in the non-replicated sequential sections. In the original program, each thread sends the reply immediately after
computing the diffs, and the replies are processed by the master thread first-come-first-serve. This way, the slave threads can compute diffs concurrently. Furthermore, the diff creation on the slave threads and the diff application on the master thread can proceed at the same time. However, with the strict ordering of multicast messages, all concurrency in the creation and the applications of diffs is lost.

5.4 Results

Our experiments are conducted on a network of 32 nodes running FreeBSD 4.1.1. Each node contains an 800MHz AMD Athlon processor and 256MB of memory. The nodes are connected by a 100Mbps switched Ethernet and a 100Mbps hub. All unicast messages go through the switch, while all multicast messages go through the hub.

Our system is based on TreadMarks 1.0.3. We compare the optimized system with the original OpenMP/TreadMarks system that uses the standard TreadMarks library. Two pointer based applications, Barnes-Hut and Ilink are used in the evaluation.

Table 5.1 and 5.3 show the execution times and the speedups of each application. We measure the execution time of the sequential program on a single node without any parallel directives. In addition to the total execution time, we also measure the time each program spent in the sequential sections. We calculate the ideal speedup based on the time the sequential program spends in the sequential sections (Ts) versus the parallel sections (Tp) of the program. With n nodes, the ideal speedup is $(Ts + Tp)/(Ts + Tp/n)$. We also measure the speedup of the parallel sections. With contention, the speedup of the parallel sections is far below the ideal value of n.

The amount of communication is shown in Tables 5.2 and 5.4. In addition to the total message and byte count, we also measure the number of messages and the amount of data sent for diff requests, and distinguish the diff requests made during the sequential sections from those made during the parallel sections. Each multicast message in the optimized version is counted as a single message. We also count the number of page faults, and the average response time for serving a page fault. These
numbers show the effect of contention reduction, as well as the multicast overhead in the replicated sequential sections. For the sequential sections, we only count the page fault number on one thread, which is the master thread in the original version, and the thread incurring the most page faults in the replicated version. For the parallel sections, we count the average number of page faults per thread. In the base system, the response time for a diff ranges from 0.16 milliseconds for a single word diff to 0.76 milliseconds for a full page one. In the optimized system, the time spent choosing a sender of a multicast request is very small, about 13 microseconds with 32 nodes.

5.4.1 Barnes-Hut

Barnes-Hut Application

Barnes-Hut [WOT+95] is an N-body simulation program using the hierarchical Barnes-Hut method. A tree structure is used to represent the recursively decomposed subdomains (cells) of the three-dimensional physical domain containing all of the particles. The leaves of the tree correspond to the particles, and are contained in a separate array.

Each iteration is divided into two steps. In the first step, a single thread reads the particles and rebuilds the tree. The second step performs force evaluation in which all threads participate. First, the threads divide the particles by traversing the tree in the Morton ordering (a linear ordering of the points in higher dimensions) of the cells. Specifically, the \( i \)th thread locates the \( i \)th segment. The size of a segment is weighted according to the workload recorded from the previous iteration. Then, each of the threads performs the force evaluation for its particles, which involves a partial traversal of the tree. During the force evaluation, a thread only modifies its own particles.

During the tree building, the master thread reads all the particles updated in the previous iteration and rewrites the tree structure. In the force evaluation phase that follows, every thread first traverses the tree to find its own particles. Contention
occurs here because everyone requests for updates to the tree simultaneously. Each thread then proceeds to update its own particles. During this period, a thread reads the tree, as well as a large part of the particles updated by the other threads during the last iteration.

In the replicated execution, the sequential tree building is replicated on all threads. Because a thread reads all the particles to build the tree, the pages containing the particles are broadcast during the replicated execution. As a result, by the end of the tree building phase, everyone caches the complete new tree and the up-to-date values of the particles.

**Barnes-Hut Results**

<table>
<thead>
<tr>
<th></th>
<th>Sequential</th>
<th>Original</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time (sec.)</td>
<td>359.4</td>
<td>53.6</td>
<td>35.5</td>
</tr>
<tr>
<td>Total Speedup</td>
<td>N/A</td>
<td>6.7</td>
<td>10.1</td>
</tr>
<tr>
<td>Sequential time (sec.)</td>
<td>1.4</td>
<td>3.2</td>
<td>14.4</td>
</tr>
<tr>
<td>Parallel time (sec.)</td>
<td>358.0</td>
<td>50.4</td>
<td>21.1</td>
</tr>
<tr>
<td>Parallel speedup</td>
<td>N/A</td>
<td>7.1</td>
<td>17.0</td>
</tr>
</tbody>
</table>

Table 5.1: Barnes-Hut. 32 nodes (Ideal speedup 28.6).

We run Barnes-Hut with 131072 particles for 2 timesteps. The sequential program runs for 359.4 seconds, of which 1.4 seconds are spent in the tree building. The ideal speedup given this sequential section is 28.6 on 32 nodes.

The speedup achieved by the base OpenMP system is 6.7 on 32 nodes, 23% of the ideal speedup. The sequential tree building takes 3.2 seconds, longer than the sequential program, because the master spends 2.1 seconds bringing in the particles from the slave threads. The parallel part takes 50.4 seconds, amounting to a speedup of 7.1 for that part alone.
<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>5,106,237</td>
<td>3,254,275</td>
</tr>
<tr>
<td></td>
<td>795,165</td>
<td>275,351</td>
</tr>
<tr>
<td>Seq</td>
<td>96,848</td>
<td>205,892</td>
</tr>
<tr>
<td></td>
<td>10,446</td>
<td>22,443</td>
</tr>
<tr>
<td></td>
<td>3,072</td>
<td>6,146</td>
</tr>
<tr>
<td></td>
<td>0.67</td>
<td>2.12</td>
</tr>
<tr>
<td>Par</td>
<td>5,006,252</td>
<td>3,045,226</td>
</tr>
<tr>
<td></td>
<td>739,139</td>
<td>221,292</td>
</tr>
<tr>
<td></td>
<td>8,479</td>
<td>3,116</td>
</tr>
<tr>
<td></td>
<td>3.34</td>
<td>0.98</td>
</tr>
</tbody>
</table>

Table 5.2: Barnes-Hut communications, 32 nodes.

The optimized OpenMP program achieves a speedup of 10.1 on 32 nodes, which is a 51% improvement over the original version, and 35% of the ideal speedup. Time spent in the parallel sections is reduced to 21.1 seconds, compared to 50.4 seconds in the base version. The number of diff messages in the parallel sections is reduced from 5,006,252 to 3,045,226, and the amount of diffs sent is reduced from 739,139 kilobytes to 221,292 kilobytes. Because the parallel sections are free of contention in the optimized version, the average response time is reduced from 3.34 milliseconds in the original version to 0.98 milliseconds. Since parallel sections are separated from the sequential sections by barriers, the execution time of the parallel section is determined by the slowest thread. In the original version, the slowest thread spends 34.6 seconds in diff requests during the parallel sections. With contention elimination and communication reduction, this time is reduced to 5 seconds in the optimized version.

The improvement is both due to the contention elimination for the tree and the broadcasting of the particles during the replicated computation. To isolate the effect of contention elimination, we hand insert broadcasting of the tree right between the
non-replicated tree building and the parallel force computation, and measure the force computation part. Without contention, time for the parallel force computation is 36.9 seconds, the number of diff messages is 4,892,246, and the amount of diffs is 538,832 kilobytes. In general, about half of the improvement is attributed to contention elimination and another half is attributed to having the particles broadcast.

The sequential part takes 14.4 seconds, 11.2 seconds longer than the base version. 0.2 seconds come from the overhead for exchanging valid notices at the beginning of the replicated execution, and the rest comes from the increased communication and the multicast overhead during the sequential sections. In the original version, the master thread has 3072 page faults, and the average response time is 0.67 milliseconds. In the replicated execution, because other threads fault on pages valid on the master thread, the number of page update requests served increases to 6146, and the average response time is increased to 2.12 milliseconds. The significant increase in average response time is largely due to the additional messages in the multicast implementation. The base system sends 96,848 messages during the sequential section, while the replicated version sends 205,892 messages, including 3,074 forwarded requests and 143,738 null acknowledgment messages. However, because all particles are broadcast here, the 11.2 second overhead in the sequential sections saves 15.8 seconds in the parallel sections that follows.

5.4.2 Ilink

Ilink Application

ILINK [CIS93, LLJO84] is a widely used genetic linkage analysis program that locates specific disease genes on chromosomes. The input to ILINK consists of several family trees. The program traverses the family trees and visits each nuclear family. The main data structure in ILINK is a pool of genarrays. A genarray contains the probability of each genotype for an individual. Since the genarray is sparse, an index array of pointers to non-zero values in the genarray is associated with each one of
them. A bank of genarrays large enough to accommodate the biggest nuclear family is allocated at the beginning of execution, and the same bank is reused for each nuclear family. When the computation moves to a new nuclear family, the pool of genarrays is reinitialized for each person in the current family. The computation either updates a parent’s genarray conditioned on the spouse and all children, or updates one child conditioned on both parents and all the other siblings.

We use the parallel algorithm described by Dwarkadas et al. [DSC+94]. Updates to each individual’s genarray are parallelized with the \texttt{parallel} directive. The bank of genarrays is shared among the threads. The master thread first examines the amount of work involved in the update, and decides to perform the update in parallel only if the amount of work exceeds a threshold. An \texttt{if} clause is used to express the conditional parallelization. A master thread then assigns the non-zero elements in the parent’s genarray to all threads in a cyclic fashion. After each thread has worked on its share of non-zero values and updated the genarray accordingly, the master thread sums up the contributions of each of the threads.

In the base OpenMP program, contention occurs because all threads come to the master for the newly initialized or updated genarrays. Contention is extremely severe when the computation moves to a new nuclear family, because the whole pool of genarrays are overwritten by the master thread, and each thread has to read genarrays of all family members in order to update one family member. In the optimized OpenMP program, the communication to fan-out the genarrays from the master thread, as well as the accompanying contention are eliminated. During the replicated execution, the contributions made by each thread during the previous iteration are broadcast to all.

\textbf{Ilink Results}

We run Ilink with the CLP input set. The sequential code runs for 99.0 seconds, with 2.2 seconds spent in the sequential sections. Because of the relatively large portion of
<table>
<thead>
<tr>
<th></th>
<th>Sequential</th>
<th>Original</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time (sec.)</td>
<td>99.0</td>
<td>53.6</td>
<td>18.0</td>
</tr>
<tr>
<td>Total Speedup</td>
<td>N/A</td>
<td>1.9</td>
<td>5.5</td>
</tr>
<tr>
<td>Sequential time (sec.)</td>
<td>2.2</td>
<td>5.5</td>
<td>9.2</td>
</tr>
<tr>
<td>Parallel time (sec.)</td>
<td>96.8</td>
<td>48.1</td>
<td>8.8</td>
</tr>
<tr>
<td>Parallel speedup</td>
<td>N/A</td>
<td>2.0</td>
<td>11.0</td>
</tr>
</tbody>
</table>

Table 5.3: IlmK. 32 nodes (Ideal speedup 18.9).

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total message data (KB)</td>
<td>1,002,787</td>
<td>230,392</td>
</tr>
<tr>
<td>Seq</td>
<td></td>
<td></td>
</tr>
<tr>
<td>diff message</td>
<td>104,530</td>
<td>94,589</td>
</tr>
<tr>
<td>diff data (KB)</td>
<td>2,803</td>
<td>2,885</td>
</tr>
<tr>
<td>page update requests</td>
<td>2,836</td>
<td>2,837</td>
</tr>
<tr>
<td>avg response time (ms)</td>
<td>0.94</td>
<td>1.71</td>
</tr>
<tr>
<td>Par</td>
<td></td>
<td></td>
</tr>
<tr>
<td>diff message</td>
<td>873,052</td>
<td>111,600</td>
</tr>
<tr>
<td>diff data (KB)</td>
<td>518,266</td>
<td>13,895</td>
</tr>
<tr>
<td>avg update requests</td>
<td>12,318</td>
<td>540</td>
</tr>
<tr>
<td>avg response time</td>
<td>3.01</td>
<td>0.64</td>
</tr>
</tbody>
</table>

Table 5.4: IlmK communications. 32 nodes.

sequential execution, the ideal speedup is 18.9 on 32 nodes. Compared with Barnes-Hut, which runs for two iterations, the IlmK execution alternates 180 times from a sequential to a parallel section, and each switch causes contention.

The base OpenMP system achieves a speedup of 1.9 on 32 nodes, 10% of the ideal speedup. The sequential part takes 5.5 seconds, of which 2.7 seconds are spent to bring in updates to the genarray from the slave threads. Because of the high rate of contention, the parallel sections take 48.1 seconds, amounting to a speedup of 2.0 for that parallel part.

The optimized OpenMP program achieves a speedup of 5.5 on 32 nodes, an 189%
improvement over the original version, and is 29% of the ideal speedup. The improve-
ment is attributed to the significant reduction in the parallel time, from 48.1 seconds
in the base version to 8.8 seconds in the optimized version. The number of diff mes-
sages in the parallel sections is reduced from 873,052 to 111,600, and the amount of
diffs sent is reduced from 518,266 kilobytes to 13,895 kilobytes, which amounts to a
87% reduction of diff requests, and a 97% reduction of diff data from the original
version. Because the parallel sections are free of contention, the average response
time is reduced from 3.01 milliseconds in the original version to 0.64 milliseconds.
In the original version, the slowest thread spends 39.8 seconds in diff requests dur-
ing the parallel sections, while this time is reduced to 0.4 seconds in the optimized
version. In contrast to Barnes-Hut, all the improvement in Ilink comes from eliminat-
ing contention for the genarrays. Because the genarrays are completely overwritten
during the sequential sections, no benefit is gained from broadcasting each thread’s
contribution to the genarray in the previous iteration.

The sequential sections take 9.2 seconds, which is 3.7 seconds longer than the
base version. Among the overhead, the program spends 1.5 second in exchanging
valid notices. This number is larger than the 0.2 seconds in Barnes-Hut, because
Ilink runs for 180 iterations instead of two, as in Barnes-Hut. The replication incurs
about the same number of page faults as the original sequential sections. However,
the average response time is increased from 0.94 milliseconds in the original version to
1.71 milliseconds. The replicated execution sends 94,589 messages versus the 104,530
messages sent in the original version. The slightly lower number of messages is the
result of combining diff requests to several threads into one multicast message. Among
the messages, 60,572 are diff requests and replies, and 33,016 are null acknowledgment
messages. Because the number of messages does not increase, the multicast overhead
in this case results mostly from the loss of concurrency in diff creation and application.
5.5 Related Work

We are not aware of any system that replicates computation in shared memory environments. Process-replication has been used in distributed systems for fault-tolerance purposes in systems such as CIRCUS [Coo85] and Manetho [EZ92]. Especially, Manetho [EZ92] depended on the fact that replicated processes executed the same deterministic program to optimize the multicast protocol. Compared with process-replication, our method only replicates parts of the program, and the replication is done for performance gains instead of for fault-tolerance.

Previous parallel computing systems using group communication spend a lot of effort on making sure the data is sent only to the threads that access it. Brazos [SB98] is a DSM system that exploits hardware multicast to improve performance. Although their results were satisfactory on a cluster of six dual-processor nodes, their method does not scale to large number of nodes. Compared to our technique, their method does not replicate the sequential computation, but instead multicasts data after global synchronization points. Without duplication, they still have to fan-out the data created at sequential execution, generating more communication than ours. Furthermore, without duplication, there are fewer threads accessing the same data. To avoid interrupting processes with too many useless multicast data, they create a different multicast group for each access pattern. Although this method works fine with the six nodes in their experiments, it does not scale. With the exponential increase of the number of access patterns, this method will quickly exceed the number of multicast groups managed by the OS, which is typically 20 in Unix. Finally, they predict access patterns according to history, and thus the benefit is limited to repetitive access patterns. We are not limited by this.

Orca [BBH+98] is a object-based distributed shared memory system built on top of totally-ordered group communication. The system requires the programmer’s extra effort to explicitly associate shared data structures with objects. Objects with low read/write ratio are stored in a single processor, while those with high read/write
ratio are replicated on all processors using multicast.

There have been several compiler techniques to generate collective communications for distributed memory systems [LC91, AL93b, AL93a, AKN95]. However, they are limited to regular array based programs. The inspector-executor method has been proposed as a way to efficiently execute irregular array based computations on distributed memory machines [SBW91]. Group communication can be applied when exchanging the data at runtime. However, the compiler analysis involved can be quite complicated [AS95, DHSK95, vHK94].

Many of the software DSM systems that support thirty-two or more processors have used SMP-based nodes [SBIS98, SGA98, SDH+97]. Thus, the actual number of nodes on the network is typically a factor of two to eight less than the number of processors. Because requests for the same page from multiple processors within a node are combined into one, the contention may not be as high as when the number of nodes in the network equals the number of processors.

Bal et al. evaluated Orca, an object-based distributed shared memory system, on a Myrinet and a Fast Ethernet network of 32 200MHz Pentium Pro computers. The object-based DSM system decreases the number of messages and data from reduced false sharing at the cost of the programmer's extra effort to explicitly associate shared data structures with objects.

Zhou et al. [ZIL96] evaluated the home-based lazy release consistency protocol against the basic LRC protocol on a 64 node Intel Paragon. The relatively large message latency, page fault, and interrupt times compared with memory and network bandwidth, and the extremely high cost of diff creation on the Paragon architecture are uncommon in modern parallel platforms and are biased towards the HLRC protocol. Our recent study [dLHL+00] on a switched, full-duplex 100 Mbps Ethernet network of thirty-two 300 MHz Pentium II-based uniprocessors showed that although the application programmer can sometime mitigate contention by carefully distributing the homes, many times, it suffers the same amount of contention as the base LRC
protocol used in TreadMarks.

5.6 Summary

Our system improves the performance of OpenMP on large scale clusters of workstations. We do so by eliminating the contention caused by sequential sections of the program. Our solution replicates the sequential computation on all threads, so that everyone modifies their locally cached copy of the shared data and does not have to go to the master thread for updates. We take advantage of the fact that all threads execute the same deterministic code to make efficient use of multicast. In addition, the fork-join execution model of OpenMP allows us to detect and replicate the sequential executions at runtime.

Our decision to multicast during the replicated execution is crucial to the scalability and flexibility of our scheme. Because each thread accesses the same set of data, a multicast message is simply sent to all threads. In addition, because all threads access the same data, no runtime speculation or compiler time analysis is necessary to predict what data to multicast. As a result, our method is not limited to regular access patterns, or repetitive accesses. We focus our experiments on two pointer based programs, Barnes-Hut and Ilp. Even with the modest scale of 32 nodes, our system achieves improvements of 51% and 189% respectively.
Chapter 6

Networks of Shared Memory Multiprocessors

Networks of shared memory multiprocessors (SMP) are a cost-effective way of building large scale DSM systems. This chapter reports on the first system that implements OpenMP on a network of shared-memory multiprocessors [HLCZ99]. This system enables the programmer to rely on a single, standard, shared-memory API for parallelization within a multiprocessor and between multiprocessors. Previously, the only standard API available on this type of platform was the message-passing interface MPI [Mes94].

In its simplest form, no change to either the TreadMarks run-time system or the OpenMP translator is necessary. The TreadMarks program can be executed on a network of SMPs by running a (Unix) process on each processor of each multiprocessor node, and having all of these processes communicate through message passing. This version, however, fails to take advantage of the hardware shared memory on the multiprocessor nodes, and optimizations to take advantage of the hardware shared memory are quite complicated [SDH+97, SBIS98]. In contrast to previous software distributed shared memory systems (SDSM) for SMPs, we modify TreadMarks to use POSIX threads for parallelism within an SMP node. This approach greatly simplifies the changes required to the SDSM in order to exploit the intra-node hardware shared memory. In addition, using the standard POSIX threads allows the implementation to be portable across different platforms.

The use of a single address space within a multiprocessor has pluses and minuses. On the positive side, it reduces the number of changes to TreadMarks to support multithreading on a multiprocessor. For example, the data within an address space
on a multiprocessor is shared by default. Furthermore, a page protection operation by
one thread applies to the other threads within the same multiprocessor; the operating
system maintains the coherence of the page mappings automatically.

On the negative side, using a single address space within a multiprocessor makes
it more difficult to provide uniform sharing of memory both between threads on
the same node and threads on different nodes. Under POSIX threads, the entire
address space is shared. Although each thread has a separate stack, a variable on one
thread’s stack can be accessed by other threads by passing them a pointer to that
variable. However, under TreadMarks, the only variables accessible to all threads
are those allocated on the shared heap. This issue is not specific to TreadMarks.
Software DSMs vary in what part of the address space is shared. In some systems,
the statically allocated variables are shared, in others the heap, in still others a special
shared memory allocation routine needs to be called to declare an area of memory as
shared. This design is a result of the high cost of tracking shared memory accesses in
software, a cost that would quickly become prohibitive if all memory is to be shared.

The OpenMP platform enables a simple approach to providing unified sharing
across the SMP cluster. The differences can be hidden by the OpenMP translator in
the same way it translates the OpenMP data environment directives to appropriate
declarations in TreadMarks. No modification to TreadMarks is necessary in order to
enable the sharing of application data or its own internal data structures within a
node. The modifications to TreadMarks is limited to separating a small amount of
thread private internal data structures, and providing mutual exclusion on accesses to
the page mapping. The OpenMP to TreadMarks translator is augmented to generate
code for the modified TreadMarks library.

We measure our OpenMP system’s performance on an IBM SP2 with multiproces-
sor nodes. The machine has four nodes, and each node has four processors. We use
seven applications: Barnes-Hut, CLU and Water from SPLASH-2 [WOT+95], 3D-
FFT from NAS [BBLS93], Red-Black SOR, TSP, and MGS. We compare the results
for our OpenMP system to two alternatives: OpenMP with the original TreadMarks, and the message passing interface MPI.

The remainder of this chapter is organized as follows. Section 6.1 presents the modifications to the original TreadMarks to support networks of SMPs. Section 6.2 describes the changes to the OpenMP translator. Section 6.3 evaluates our system's overall performance and compares it to the original TreadMarks and to MPI. Section 6.4 discusses related work. Section 6.5 concludes this chapter.

6.1 Modifications to TreadMarks

TreadMarks is modified to use POSIX threads to implement parallelism within a multiprocessor node. By using POSIX threads, the OpenMP threads within a multiprocessor share a single address space, and coherence is maintained automatically by the hardware. Thus, we do not have to modify TreadMarks to enable the sharing of application data or its own internal data structures between threads within the same node. We do, however, have to modify TreadMarks to place some data structures, such as message buffers, in thread-private memory.

Synchronizing accesses to TreadMarks' shared internal data structures by the threads within a node is straightforward. TreadMarks already guards accesses to critical data structures by making sure that only one event at a time, either a page fault, a local call to TreadMarks, or an incoming request, will access a critical data structure. Thus, we simply change the existing synchronization to work with POSIX threads with one exception, where we add a per-page mutex to allow greater concurrency in the page fault handler.

Since page mappings are shared among the threads on a node, we change the memory mappings to make sure that another thread cannot access a page until a thread finishes updating it. Similar to Millipede [IS99], we add a second mapping at a different address within each node's address space for the memory shared between nodes. The first, or original, mapping is used exclusively by the application. The
TreadMarks code uses the second mapping, which permits read and write access at all times, to update shared memory pages so that the application’s mapping can remain invalid while the update is in progress.

In fact, the use of two mappings reduces the number of mprotect, or page protection, operations performed by TreadMarks, even on a single-processor node. For example, in the original TreadMarks, a read access to an invalid page would result in two mprotect operations: one to enable write access in order to update the page and another to make the page read-only after the update. In the modified version, only the latter mprotect operation is performed. The second mapping eliminates the need for the first mprotect operation.

Our system performs much fewer mprotect calls than systems that use Unix processes such as Cashmere-2L [SDH+97]. With POSIX threads, an mprotect by one thread applies to the other threads within the same node because they share the same address space. With Unix processes on the other hand, the same mprotect must be performed in each process’s address space even if the underlying memory is shared between address spaces.

Finally, the synchronization functions provided by TreadMarks to the application are modified to combine the use of POSIX threads-based synchronization within a node and the existing TreadMarks implementation between nodes. Thus, the program can continue to use a single API, that of TreadMarks, for synchronization.

6.2 Modifications to Translator

In our modified version of TreadMarks, global variables are shared between threads within a multiprocessor but are private with respect to threads on a different multiprocessor. Rather than attempting to solve this problem in the run-time system, we chose to address it in the OpenMP translator where a solution is straightforward.

In OpenMP, global variables are either shared, private or threadprivate. The original OpenMP translator allocates all shared global variables on the shared heap,
thus making sure they are shared across nodes. Private variables are redeclared within the parallel subroutine, so that each thread has its own copy on the stack. The original OpenMP translator for TreadMarks leaves threadprivate variables in place. As a result, they are shared within a node, but private across nodes. To solve this problem, we change the translator to allocate an array of $n$ copies of the threadprivate variable, where $n$ is the number of threads per node. Each reference to the global variable is replaced by a reference to the array, specifically, a reference to the element corresponding to the thread’s (local) id.

6.3 Results

We use seven applications in this study: SPLASH-2 Barnes-Hut, NAS 3D-FFT, SPLASH-2 CLU and Water, Red-Black SOR, TSP and MGS. See Chapter 3 for discussions of these applications. Table 6.1 summarizes the problem size and the running time of the sequential programs. The sequential running times are used as the basis for calculating speedups.

Our experimental platform is an IBM SP2 consisting of four SMP nodes. Each node contains four IBM PowerPC 604 processors and 1 Gbyte of memory. All of the nodes are running AIX 4.2.

<table>
<thead>
<tr>
<th>Appl.</th>
<th>Size, Iterations</th>
<th>Sequential Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>65536</td>
<td>158.0</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>128×128×64, 10</td>
<td>65.2</td>
</tr>
<tr>
<td>CLU</td>
<td>2048×2048, block: 32</td>
<td>86.9</td>
</tr>
<tr>
<td>Water</td>
<td>4096, 4</td>
<td>760.3</td>
</tr>
<tr>
<td>SOR</td>
<td>8K x 4K, 20</td>
<td>149.0</td>
</tr>
<tr>
<td>TSP</td>
<td>19 cities, -r14</td>
<td>248.1</td>
</tr>
<tr>
<td>MGS</td>
<td>2K x 2K</td>
<td>563.3</td>
</tr>
</tbody>
</table>

Table 6.1 : Problem size and sequential execution time.

We first compare the performance of the OpenMP programs translated to run on
Figure 6.1: Speedups on four four-processor nodes.

the modified TreadMarks library using POSIX threads (OpenMP/thread) against the performance of those same programs translated into original TreadMarks programs using processes (OpenMP/original). In the OpenMP/original, processes on the same node communicate via message passing instead of using the hardware shared memory.

We then compare the OpenMP/thread and OpenMP/original versions of the applications against MPI versions of the same applications. We use the MPICH (www.mcs.anl.gov/mpi/mpich) implementation of MPI, because it takes advantage of the hardware shared memory when sending messages within the same node. For the MPI programs, we count both the total number of messages and the number of messages that actually cross node boundaries.

Figure 6.1 shows the speedups for the OpenMP/original programs with four processes per node, OpenMP/thread programs with four threads per node, and MPI programs with four processes per node on the four-node SP2. Table 6.2 compares the amount of data and the number of messages communicated for the different cases.
<table>
<thead>
<tr>
<th>Appl.</th>
<th>OpenMP/ original</th>
<th>OpenMP/ thread</th>
<th>MPI Total</th>
<th>Off-node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data (Mbytes)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Barnes</td>
<td>593.3</td>
<td>196.5</td>
<td>259.7</td>
<td>207.8</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>159.4</td>
<td>126.5</td>
<td>157.3</td>
<td>125.8</td>
</tr>
<tr>
<td>CLU</td>
<td>102.2</td>
<td>51.2</td>
<td>102.2</td>
<td>51.1</td>
</tr>
<tr>
<td>Water</td>
<td>192.3</td>
<td>42.7</td>
<td>34.6</td>
<td>26.0</td>
</tr>
<tr>
<td>SOR</td>
<td>0.64</td>
<td>0.07</td>
<td>9.8</td>
<td>2.0</td>
</tr>
<tr>
<td>TSP</td>
<td>2.8</td>
<td>0.55</td>
<td>0.03</td>
<td>0.026</td>
</tr>
<tr>
<td>MGS</td>
<td>508.6</td>
<td>102.2</td>
<td>251.6</td>
<td>201.3</td>
</tr>
<tr>
<td>Messages</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Barnes</td>
<td>1478908</td>
<td>156371</td>
<td>720</td>
<td>576</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>40975</td>
<td>31694</td>
<td>9750</td>
<td>7800</td>
</tr>
<tr>
<td>CLU</td>
<td>28895</td>
<td>13239</td>
<td>1860</td>
<td>930</td>
</tr>
<tr>
<td>Water</td>
<td>78402</td>
<td>24067</td>
<td>1776</td>
<td>1344</td>
</tr>
<tr>
<td>SOR</td>
<td>3637</td>
<td>735</td>
<td>1200</td>
<td>240</td>
</tr>
<tr>
<td>TSP</td>
<td>9227</td>
<td>4853</td>
<td>1256</td>
<td>1070</td>
</tr>
<tr>
<td>MGS</td>
<td>184583</td>
<td>37041</td>
<td>30720</td>
<td>24576</td>
</tr>
</tbody>
</table>

Table 6.2 : Amount of data and number of messages on four four-processor nodes.

6.3.1 OpenMP/original versus OpenMP/thread

The improvement of OpenMP/thread over OpenMP/original mainly comes from the reduction of communication. Overall, compared to OpenMP/original, OpenMP/thread sends less data, from a low of 21% less data for 3D-FFT to a high of 9.1 times less data for SOR, and fewer messages, from a low of 24% fewer messages for 3D-FFT to a high of 9.5 times fewer messages for Barnes. As a result, OpenMP/thread achieves up to 30% better speedups than the latter for all applications except 3-D FFT, for which the thread version is 8% slower than the original version.

In terms of relative speedups, the applications can be categorized into three groups. The first group, consisting of CLU, TSP, and MGS, has low to moderate computation to communication ratios. For these programs, the 2 to 5 fold reduction in the amount of data transmitted results in significant speedups. The second
group, consisting of Barnes, Water, and SOR, has high computation to communication ratios. In this case, a 3 to 9 fold reduction in data leads to little improvement in running time. FFT forms the third group, where there is only a 21% reduction of data communicated, and we see a slight slowdown for the OpenMP/thread code. This is where a secondary factor, namely the cost of page protection operations affects the performance, as will be explained later.

Table 6.3 compares the number of page protects, the number of page faults, and the number of diffs in the original and the thread versions of OpenMP programs. It presents the results of using all four-processor on each of the four SMP nodes, as well as that of using only one processor per node.

First, the OpenMP/thread programs with one thread per node perform 25% to 56% fewer mprotect operations than the corresponding OpenMP/original versions with one process per node, indicating that the alias mapping (see Section 6.1) reduces the number of mprotect operations independent of any multithreading effects. Second, the OpenMP/thread programs with four threads per node perform 1.9 to 6.2 times fewer mprotect operations than the OpenMP/original codes with four processes per node.

To further separate the contributions of double mapping and multithreading, we disable multithreading in the OpenMP/thread version and count the number of mprotects. Table 6.4 shows the mprotect counts of OpenMP/thread with 16 processes on four four-processor nodes. The comparison of these numbers with the mprotect counts in Table 6.3, shows that the use of double mapping alone reduces the number of mprotects by 1.2 to 1.8 times, and the use of multithreading reduces the number of mprotects by an additional 1.5 to 5 times.

Multithreading also reduces the number of page faults. With one processor per node, the number of page faults incurred by OpenMP/thread is the same as that of OpenMP/original, but when running on all four processors on each node, OpenMP/thread programs with four threads per node incur 1.2 to 5 times fewer page
<table>
<thead>
<tr>
<th>Application</th>
<th>Orig/1</th>
<th>Thrd/1</th>
<th>Orig/4</th>
<th>Thrd/4</th>
</tr>
</thead>
<tbody>
<tr>
<td>mprotect Count</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Barnes</td>
<td>112452</td>
<td>73322</td>
<td>320743</td>
<td>107114</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>65650</td>
<td>50200</td>
<td>97690</td>
<td>50588</td>
</tr>
<tr>
<td>CLU</td>
<td>12411</td>
<td>8253</td>
<td>29015</td>
<td>8308</td>
</tr>
<tr>
<td>Water</td>
<td>27244</td>
<td>19071</td>
<td>102879</td>
<td>27691</td>
</tr>
<tr>
<td>SOR</td>
<td>1209</td>
<td>969</td>
<td>6037</td>
<td>969</td>
</tr>
<tr>
<td>TSP</td>
<td>6947</td>
<td>5529</td>
<td>11628</td>
<td>5438</td>
</tr>
<tr>
<td>MGS</td>
<td>30730</td>
<td>21511</td>
<td>86154</td>
<td>21118</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Fault Count</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>55810</td>
<td>55810</td>
<td>161565</td>
<td>83349</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>30860</td>
<td>30860</td>
<td>39020</td>
<td>31155</td>
</tr>
<tr>
<td>CLU</td>
<td>4158</td>
<td>4158</td>
<td>12460</td>
<td>6230</td>
</tr>
<tr>
<td>Water</td>
<td>13533</td>
<td>13523</td>
<td>46130</td>
<td>28705</td>
</tr>
<tr>
<td>SOR</td>
<td>480</td>
<td>480</td>
<td>2400</td>
<td>480</td>
</tr>
<tr>
<td>TSP</td>
<td>2895</td>
<td>2889</td>
<td>4794</td>
<td>4047</td>
</tr>
<tr>
<td>MGS</td>
<td>14336</td>
<td>14336</td>
<td>40346</td>
<td>32404</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Diff Count</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>29941</td>
<td>29941</td>
<td>87651</td>
<td>44333</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>15404</td>
<td>15404</td>
<td>19370</td>
<td>15501</td>
</tr>
<tr>
<td>CLU</td>
<td>4095</td>
<td>4095</td>
<td>4095</td>
<td>2079</td>
</tr>
<tr>
<td>Water</td>
<td>5090</td>
<td>5090</td>
<td>13017</td>
<td>7890</td>
</tr>
<tr>
<td>SOR</td>
<td>240</td>
<td>240</td>
<td>1200</td>
<td>240</td>
</tr>
<tr>
<td>TSP</td>
<td>1394</td>
<td>1394</td>
<td>1599</td>
<td>1357</td>
</tr>
<tr>
<td>MGS</td>
<td>3072</td>
<td>3072</td>
<td>3827</td>
<td>3724</td>
</tr>
</tbody>
</table>

Table 6.3: Number of mprotects, page faults, and diffs on four SMP nodes. 1 – use only one processor per node. 4 – use four processors per node.
<table>
<thead>
<tr>
<th>Application</th>
<th>mprotect Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>207293</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>78040</td>
</tr>
<tr>
<td>CLU</td>
<td>16555</td>
</tr>
<tr>
<td>Water</td>
<td>60667</td>
</tr>
<tr>
<td>SOR</td>
<td>4840</td>
</tr>
<tr>
<td>TSP</td>
<td>8211</td>
</tr>
<tr>
<td>MGS</td>
<td>48313</td>
</tr>
</tbody>
</table>

Table 6.4: mprotect counts of OpenMP/thread with 16 processes on four four-processor nodes. One thread per process.

faults than their OpenMP/original counterparts with four processes per node. This reduction comes from two sources. First, multithreading relies on hardware to keep memory consistent within a node, thus eliminates the page fault incurred by ThreadMarks to keep consistency between processes within a node. Second, when multiple threads access the same page, only the first one has to pay a page fault if the page is protected, whereas each of the processes on a node has to fault once. For the same reason that hardware keeps pages consistent within a node, multithreading reduces the number of diffs by 3% to 80% from OpenMP/original.

To identify the exact reason for the slowdown of the OpenMP/thread version of 3-D FFT, we measure the average time of an mprotect in both the thread version and the process version. The results are presented in Table 6.5. With four threads per node, the average cost of an mprotect is 1.8 times to 10.3 times higher than that of OpenMP/process. For 3-D FFT, the average cost of an mprotect increases by a factor of ten, while the number of mprotects decreases by less than half. As a result, the time spent on mprotects increases by a factor of five. This, coupled with the small reduction in communication, explains the slight slowdown in the OpenMP/thread version of 3-D FFT.

The increase in mprotect cost is a result of the way AIX 4.2 implements virtual memory. The operating system keeps a linked list of allocated pages in the address
<table>
<thead>
<tr>
<th>Application</th>
<th>Original</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>65.6</td>
<td>115.5</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>62.5</td>
<td>644.9</td>
</tr>
<tr>
<td>CLU</td>
<td>61.8</td>
<td>555.5</td>
</tr>
<tr>
<td>Water</td>
<td>72.2</td>
<td>128.4</td>
</tr>
<tr>
<td>SOR</td>
<td>142.5</td>
<td>708.5</td>
</tr>
<tr>
<td>TSP</td>
<td>42.7</td>
<td>85.2</td>
</tr>
<tr>
<td>MGS</td>
<td>67.5</td>
<td>234.8</td>
</tr>
</tbody>
</table>

Table 6.5: Average `mprotect` time (usec) on four four-processor nodes.

space. To find a page entry in the list, it traverses the list starting from the entry for the last page accessed. This design is meant to take advantage of locality, but in our multithreaded parallel programs, different threads tend to access distinct regions of the data, thus resulting in long searches through the list.

6.3.2 OpenMP versus MPI

A previous study comparing SDSM with message passing [LDCZ97] has shown that, in general, SDSM programs send more messages and data than message passing versions due to the separation of synchronization and data transfer, the need to handle access misses caused by the use of an invalidate protocol, false sharing, and diff accumulation for migratory data. In our experiments, the performance difference ranges from 7% for 3-D FFT, to 30% for Barnes. The OpenMP/original programs sent between 5.3 and 2568 times more messages than their MPI counterparts, for which only off-node messages are counted. The reason that the MPI version of Barnes sends so many fewer messages is that it replicates the particles and duplicates the tree building in every process. As a consequence, within an iteration, the only communication by each process is a single broadcast of all the particles modified by that process. Except for SOR, the amount of data sent by OpenMP/original ranges from 1.27 times more for 3D-FFT to 93 times more for TSP, compared with off-node data in the MPI programs.
For SOR, because a large percentage of the elements remain unchanged, and because TreadMarks only communicates diffs, the OpenMP program sends 15.5 times less data than the MPI code, which always communicates whole boundary rows. 

Our results with the OpenMP/thread programs show that on SMP nodes using multithreading can significantly reduce the gaps in the number of messages and the amount of data transmitted between SDSM and MPI programs. Compared to the off-node communications of MPI, the OpenMP/thread programs send from 1.5 times more messages, for MGS, to only 271 times more messages, for Barnes. Similarly, OpenMP/thread sends 2 to 28.6 times less data than MPI for two out of the seven applications, about the same amount of data for three others, and only 1.6 to 21.2 times more data than MPI for the remaining two.

Table 6.2 further shows that for all applications except SOR and CLU, the MPI versions send about 12/15 of the total data and messages across node boundaries. This corresponds to the ratio of off-node processors versus all processors as viewed by each processor. For SOR, the MPI program sends only 20\% of the total data and messages across node boundaries because communication only occurs between neighboring processes and the neighboring processes are in most cases within the same node. For CLU, the MPI program sends half of the data and messages between nodes and half within nodes. This is because the four processors in charge of each row of blocks are in the same node and the four processors in charge of each column of blocks are across four nodes.

6.4 Related Work

We are aware of five implementations of SDSM on networks of SMPs [ENCH96, SBIS98, SGA98, SB97, SDH+97]. Compared to our solution, none of them use the standard POSIX threads within an SMP node. SoftFLASH [ENCH96] implements the software DSM in the Power Challenge Irix kernel. Compared with their approach, our using of POSIX threads is relatively portable. The Shasta system [SGA98] supports
fine-grain shared memory across SMP nodes. The systems takes the binary code for a hardware shared memory machine, and inserts in-line checking to check the software shared memory state before each shared memory load or store. Novel techniques are developed to minimize the overhead of in-line checks. Because the system requires instrumenting the binary code, significant effort is needed to port it from one hardware platform to another. Cashmere-2L [SDH97] uses Unix processes instead of POSIX threads. Therefore, it must perform the same mprotect in each process’s address space. The reason is that mprotect only applies to the calling process’s address space, even if the underlying memory is shared between address spaces. The protocol implemented by Cashmere-2L also takes advantage of the Memory Channel network interface unique to the DEC Alpha machines. Samanta et al. [SBIS98] present an implementation of a lazy, home-based, multiple-writer protocol across SMP nodes. Similar to Cashmere-2L, their implementation uses Unix processes instead of POSIX threads. Similar to the multithreaded TreadMarks, the Brazos system [SB97] also uses multiple threads to take advantage of the hardware shared memory within each SMP node on a cluster of SMP nodes running Windows NT. However, the integration of the two-level consistency protocols (within and across SMP nodes) is not explicitly addressed.

6.5 Summary

This chapter presents a system that implements OpenMP on a network of shared memory multiprocessors. This system enables the programmer to rely on a single, standard, shared memory API for parallelization within a multiprocessor and between multiprocessors. It exploits the hardware shared memory within an SMP node using the standard POSIX threads.

Using the hardware shared memory within an SMP node can significantly reduce data and messages transmitted by a SDSM. For our collection of applications, the version using threads within a node send from a low of 21% less data and 24% fewer
messages to a high of 9.1 times less data and 8.4 times fewer messages than those using processes within a node. As a consequence, they achieve up to 30% better speedups than the latter for all applications except 3D-FFT, for which the thread version is 8% slower than the original version. The slowdown in 3D-FFT is due to inefficient implementation of page protection operations in AIX 4.2. Overall, the speedups of multithreaded TreadMarks codes on four four-way SMP SP2 nodes are within 7% to 30% of the MPI versions.
Chapter 7

Transparent Adaptive Parallelism

Networks of workstations (NOWs) differ from dedicated computing environments in that individual nodes in a NOW become available or unavailable as the workstation owner goes away or returns. To be truly useful, a parallel processing system for a NOW must be able to adapt to a continually changing pool of available nodes. Ideally, this adaptation should be transparent, allowing the user to program in a relatively standard way, without requiring any special-purpose code in the application. Such an adaptive parallel processing system is also useful in other environments (e.g., a parallel computer with a changing job mix), but in this chapter we focus on a NOW, because there adaptivity is a requirement, not just an added feature.

Recent parallel programming models like HPF [KLS+94] or OpenMP [Ope98] shelter the user from having to deal with some aspects of parallel programming, such as the number of nodes, the low-level details of iteration or data partitioning, or the communication of data between nodes. These properties simplify parallel programming, but, in addition, they also provide the foundation for transparent adaptive parallelism. Since aspects like the number of nodes are handled by the system and not the user, it becomes possible to change them adaptively without user intervention.

In an OpenMP program, the programmer specifies, roughly speaking, what pieces of the code can be run in parallel. The number of threads executing these parallel constructs need not be hardwired. Therefore, adjusting the number of threads at runtime can be done transparently. Furthermore, OpenMP’s execution model, consisting of a succession of sequential code and parallel constructs, naturally suggests efficient adaptation points at the beginning or the end of these parallel constructs.
One of the main technical challenges in supporting adaptivity on a NOW is to transparently move application data around at the time of adaptation, either moving it to newly joining nodes or moving it off leaving nodes. We rely on a software distributed shared memory (SDSM) runtime system [LH89] to avoid any need for user intervention in this task. Automatic data distribution is the main advantage of such systems, regardless of adaptivity. Here, that same feature is used to support automatic data re-distribution after an adaptation has taken place.

Our adaptive system is an extension of the TreadMarks SDSM system [ACD+96], and uses the OpenMP translator explained in Chapter 2 to generate TreadMarks code from OpenMP programs [LHZ98]. No code is added to the application specifically to obtain adaptivity. We demonstrate the performance of our system for different rates of adaptation and compare the results with non-adaptive runs of the same applications on the non-adaptive base TreadMarks system. We analyze the factors contributing to the cost of node joins and leaves. We conclude that for moderate rates of adaptation, the cost of adaptation is well within acceptable range and is a cost well worth paying for the added flexibility and functionality.

The outline of the rest of this chapter is as follows. Section 7.1 demonstrates how transparent adaptation is achieved. Section 7.2 describes the implementation of the system, and Section 7.3 assesses its performance. Related work is covered in Section 7.4, and Section 7.5 offers some discussion and concluding remarks.

### 7.1 Transparent adaptation

We present the added functionality for transparent adaptation in this section; the implementation is discussed in Section 7.2.

Each node normally executes one thread. When a new node becomes available and starts participating in a computation, this is called a *join event*. When a node withdraws, we speak of a *leave event*. An *adapt event* is either a join or a leave event.

A request for an adapt event may occur anytime, but it is usually only executed
at the next *adaptation point*. OpenMP provides natural adaptation points at the beginning or the end of the execution of a parallel construct. At these points, joins and leaves can be handled efficiently by increasing or decreasing the number of threads and re-partitioning the loop iterations among them.

Join events have the nice property that the system can always delay its response. If the event arrives while the system is not at an adaptation point, the system simply ignores the availability of an extra node until the computation reaches the next adaptation point. We call this behavior a (normal) *join*; Figure 7.1.a depicts such a situation.

Leave events are more complicated to handle since presumably the workstation node is needed for some other task with a higher-priority. For a leave event, if the computation can reach the next adaptation point within a specifiable time limit, termed the *grace period*, we let the leave events take effect there. In this case, handling the leave event is greatly simplified since it is processed at a time when the system is free to determine the number of threads. We call this a *normal leave*. Figure 7.1.b depicts this case: a leaving thread reaches an adaptation point within the grace period and is terminated at the adaptation point. For a normal leave, only the data that are still needed by other threads need to be moved.

![Diagram](image)

Figure 7.1: (Normal) join (a), normal leave (b), and urgent leave (c).
If the computation does not reach an adaptation point within the grace period, then the current thread is migrated to another node in the system. The thread is then executed on that node by multiplexing it with the thread already running on that node until the next adaptation point is reached. At that time, processing proceeds as in the case of a normal leave. We call this sequence an *urgent leave*, it is depicted in Figure 7.1.c. Urgent leaves cause much more data to be moved than normal leaves, because all intermediate data of the migrating thread needs to be transferred. In addition, if a computation is balanced for $t$ threads, multiplexing one node may idle the $t - 2$ non-multiplexed nodes for some time.

Since adaptation points are reached fairly frequently (in many applications several adaptation points are reached per second), urgent leaves are typically not needed. The concept of a grace period allows fine tuning the management of the workstation nodes, since this period can be node-specific (and may even vary during a day). The owner of a workstation node is not denied service during the grace period. The node is, however, shared with the computation that occupies a node.

Although all of the foregoing describes the joining or leaving of a single node, the system also supports multiple adaptations. In fact, one of the additional advantages of postponing adaptations is that many of them can be handled at once, leading to reduced overhead. Finally, the system also provides checkpointing to recover from catastrophic failures such as a crash, power flicker, or a machine reboot.

### 7.2 Implementation of the adaptive system

In this section we elaborate on the support added to the base TreadMarks system described in Chapter 2 to implement adaptation. These changes are purely TreadMarks-internal, without *any* changes to the TreadMarks API or the operating system, so we can run standard TreadMarks programs, and we can convert OpenMP programs to TreadMarks programs using the same compiler we use for the non-adaptive version (See Chapter 2).
Each node recognizes join and leave events and communicates those to the master. How these events are generated is beyond the scope of this paper. E.g., a daemon may generate events at set times according to an operational schedule, or a load sensor may be employed to make load-dependent decisions.

For simplicity, most of the following discussion focuses on the implementation of a single join event or a single leave event. The system is, however, capable of dealing with multiple join and leave events at a single adaptation point.

The principal implementation challenge is to make sure that after an adaptation memory continues to be shared consistently between the remaining threads. In other words, any shared memory information residing solely on leaving threads, whether shared memory data or shared memory state information, must be recorded in one or more of the remaining threads. Furthermore, any newly joining threads must receive the necessary shared memory state information so that they can locate the data that they need to access later.

To simplify this task, a central idea of our implementation is to use the garbage collection mechanism, already present in the non-adaptive TreadMarks system. During execution, both the non-adaptive and the adaptive version of TreadMarks accumulate a variety of consistency information, primarily twins, diffs, and write notices [ACD+96]. When the memory allocated for these data structures becomes exhausted, TreadMarks initiates a garbage collection, typically at a barrier or at a Tmkfork. The garbage collection removes all these internal data structures, and leaves each memory page either valid and up-to-date, or invalid but with its “owner field” pointing to a node with a valid copy of the page. By executing a garbage collection when an adaptation event takes place, we considerably reduce the complexity of adaptation and the amount of data that needs to be exchanged. In particular, no consistency data needs to be sent at the time of an adaptation.
7.2.1 Join events

The master spawns a new thread on the specified host. While all threads continue normally, the new thread asynchronously sets up network connections, first to all other slave threads, then to the master. Therefore, when the master receives this connection request, it knows that the new thread has set up all its other connections and is ready to join the computation.

When all current threads have arrived at the adaptation point, the master initiates a garbage collection. Afterwards, the master sends the joining thread a message describing where an up-to-date copy of every shared memory page is located.

The thread identifiers are (re-)assigned, and the total number of threads is reset. Then the master sends the next `mk_fork` message to the new set of threads, and each thread determines a (new) iteration partitioning based on its thread identifier and the total number of threads, using the code generated by the OpenMP compiler.

Such a re-partitioning of the iteration space causes in many cases a re-distribution of the data. This re-distribution, if any, happens during further execution of the application, as a result of threads fetching pages on a page fault, using the normal SDSM mechanisms.

7.2.2 Normal leave events

The handling of normal leave events is similar to the handling of join events. When all threads have reached the adaptation point, the master initiates a garbage collection. It then suffices to move all pages exclusively valid at the leaving thread elsewhere. To expedite this transfer, each of the remaining threads is sent a roughly equal number of these pages, allowing the data to be sent in parallel. Furthermore, several pages are sent in a single message, further reducing the cost. The master then informs all threads of the identity of the new owners of these pages. The remainder of the leave procedure is identical to that for a join, including re-assignment of thread identifiers, the `mk_fork` message sent by the master, and any data re-distribution.
As an additional optimization, if an equal number of threads join and leave the computation at the same adaptation point, the joining threads try to "take the place" of the leaving threads: the pages exclusively valid at a leaving thread are sent immediately to a newly joining thread.

7.2.3 Urgent leave events

When a thread needs to migrate to another host, a new thread is first created on that host, and the interprocessor communication connections are set up to this thread. All threads then wait for the completion of the migration. We rely on a modified version of the libckpt library to implement migration [PBKL95]. Libckpt is designed for checkpointing to disk and for recovery from that checkpoint, but we have modified it to write out the heap and the stack of the leaving thread to the newly created thread, and then start that thread.

7.2.4 Fault tolerance

We use checkpointing for fault tolerance and include a brief description of it here, because fault tolerance is a natural complement to adaptivity.

Whereas a distributed computation normally requires a consistent checkpoint [CL85], or some form of message logging [JZ90], to guarantee correct recovery, we can avoid much of this complication by limiting checkpoints to the OpenMP adaptation points. At these points in the execution, the slave threads do not have any private "thread" state (such as a stack) that needs to be recovered; they only have shared memory state. Only the master thread has thread state that needs to be recovered.

Checkpointing is therefore done periodically but only at an adaptation point. First, a garbage collection is invoked to bring shared memory into a well-defined state. Second, the master collects all pages for which it does not have a valid copy. Finally, the master uses the libckpt library to checkpoint itself to disk. No checkpointing by slave threads is required, avoiding the considerable complexity of checkpoint and
7.2.5 Current limitations

The master thread can migrate but it currently cannot perform a normal leave.

Adaptivity is limited to programs that repeatedly fork and join. Applications that fork once at the beginning and join once at the end run non-adaptively.

It should also be understood that it is quite possible in OpenMP for the user to explicitly code the iteration partitioning in terms of the thread identifiers and the number of threads. Clearly, adaptivity will not have any benefit for such applications. It is also possible for the user to explicitly disable adaptivity by setting the switch that OpenMP provides for this purpose.

7.3 Performance

We use a switched, full-duplex 100Mbps Ethernet network, connecting 8 300Mhz Pentium II machines. Each machine has a 512K bytes secondary cache and 256M bytes of memory. The machines run FreeBSD 2.2.6, and use UDP sockets to communicate with each other. The roundtrip latency for a 1-byte message is 126 microseconds. The time to acquire a lock varies between 178 and 272 microseconds. An 8-node barrier takes 359 microseconds. The time for getting a diff varies between 313 and 1,544 microseconds, depending on the size of the diff. A full page transfer takes 1,308 microseconds.

We use six applications to assess the performance of our adaptive SDSM system, including both regular and irregular applications. Jacobi and Modified Gramm-Schmidt (MGS) are simple numerical codes. 3D-FFT comes from the NAS benchmark suite. NBF (Non-Bonded Force) is the kernel of a molecular dynamics programs. Water and Barnes are two applications from the SPLASH-2 benchmark suite. Table 7.1 lists the problem size, the amount of shared memory consumed, and the sequential time of each program. Chapter 3 provides a detailed discussion of these programs.
<table>
<thead>
<tr>
<th></th>
<th>Program Size</th>
<th>Iterations</th>
<th>Share Memory Size</th>
<th>Sequential time(sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MGS</td>
<td>3072 x 3072</td>
<td>3072</td>
<td>48 MB</td>
<td>1404.20</td>
</tr>
<tr>
<td>Jacobi</td>
<td>2500 x 2500</td>
<td>1000</td>
<td>47.8 MB</td>
<td>1283.63</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>128 x 64 x 64</td>
<td>100</td>
<td>42 MB</td>
<td>280.90</td>
</tr>
<tr>
<td>NBF</td>
<td>131072 atoms, 80 partners</td>
<td>100</td>
<td>52MB</td>
<td>2398.79</td>
</tr>
<tr>
<td>Barnes-Hut</td>
<td>32768 bodies</td>
<td>20</td>
<td>6 MB</td>
<td>479.18</td>
</tr>
<tr>
<td>Water</td>
<td>1728 molecules</td>
<td>20</td>
<td>1.5 MB</td>
<td>759.83</td>
</tr>
</tbody>
</table>

Table 7.1: Problem size and sequential time of applications.

### 7.3.1 Key overall results

In this section we discuss the cost of join and leave events. In the next section we analyze these costs based on detailed implementation measurements.

**Overhead for providing adaptivity**

In the absence of adapt events, there is no cost to supporting adaptivity compared to the non-adaptive base system. Table 7.2 compares the execution times for the non-adaptive TreadMarks system and for our adaptive system without any adaptations. The results demonstrate that in the absence of adapt events the overhead of the adaptive system is virtually nil. The slight performance gain of our system in some cases is coincidental. More importantly, the network traffic is identical in both systems.

**Cost of normal joins and leaves**

It is not straightforward to quantify the cost of adaptation. The cost of an adapt event varies, even for a single application. For instance, the thread id of the leaving thread may significantly affect the amount of data to be moved, as demonstrated by the schematic example in Figure 7.2. Many other factors affect the cost of joins and leaves, as documented in Section 7.3.2.
<table>
<thead>
<tr>
<th>Nodes</th>
<th>Time(seconds)</th>
<th>Number/amount of transfers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Standard</td>
<td>Adaptive</td>
</tr>
<tr>
<td>MGS</td>
<td>8  243.46</td>
<td>412.14</td>
</tr>
<tr>
<td></td>
<td>4  398.07</td>
<td>397.23</td>
</tr>
<tr>
<td>Jacobi</td>
<td>8  215.06</td>
<td>216.17</td>
</tr>
<tr>
<td></td>
<td>4  361.38</td>
<td>362.88</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>8  83.50</td>
<td>81.95</td>
</tr>
<tr>
<td></td>
<td>4  138.20</td>
<td>133.51</td>
</tr>
<tr>
<td>NBF</td>
<td>8  535.89</td>
<td>534.74</td>
</tr>
<tr>
<td></td>
<td>4  714.78</td>
<td>715.36</td>
</tr>
<tr>
<td>Barnes</td>
<td>8  96.18</td>
<td>96.92</td>
</tr>
<tr>
<td></td>
<td>4  150.76</td>
<td>152.93</td>
</tr>
<tr>
<td>Water</td>
<td>8  123.81</td>
<td>123.32</td>
</tr>
<tr>
<td></td>
<td>4  232.97</td>
<td>235.30</td>
</tr>
</tbody>
</table>

Table 7.2: Execution times and network traffic on the non-adaptive system and the adaptive system with no adapt events. Network traffic is identical on both systems.

To provide an idea of the overhead of adaptation, we periodically cause an adapt event to occur during the execution of an application. We report results for a range of periods at which adapt events occur. Figure 7.3 shows the results for each application. In these experiments, the grace period is made large enough such that all adapt events can occur at adaptation points. The application starts out with 4 or 8 threads. Alternately, at the end of a period, a thread leaves, or a thread joins. The leaving thread is always the one with the highest thread id, exhibiting a worst-case for many applications (see Figure 7.2). The x-axis shows the period at which adapt events occur. The y-axis shows the execution time. The figures demonstrate that, for these applications, the system can support up to several adaptations per minute with only modest performance loss.

Tables 7.3 provides a different view of the results. To obtain these results, a single adapt event was made to occur during the execution of an application. Again, in the case of a leave, the last thread was made to leave. To produce a representative result, we performed this experiment a number of times, changing the time at which
Figure 7.2: Effect of thread id of leaving node on the amount of data moved at adaptation time, with a simple block distribution of the data among threads. The shaded areas indicate the data to be moved. Up to 50% of the data space is moved if node 7 leaves (a), only up to 30% if node 3 leaves.

<table>
<thead>
<tr>
<th>Application</th>
<th>Normal Leave</th>
<th>Normal Join</th>
<th>Migration</th>
</tr>
</thead>
<tbody>
<tr>
<td>MGS</td>
<td>1.93</td>
<td>2.30</td>
<td>6.90</td>
</tr>
<tr>
<td>Jacobi</td>
<td>2.20</td>
<td>1.73</td>
<td>6.70</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>1.30</td>
<td>1.16</td>
<td>6.13</td>
</tr>
<tr>
<td>NBF</td>
<td>1.32</td>
<td>0.48</td>
<td>7.66</td>
</tr>
<tr>
<td>Barnes</td>
<td>2.46</td>
<td>1.08</td>
<td>1.75</td>
</tr>
<tr>
<td>Water</td>
<td>0.08</td>
<td>0.01</td>
<td>0.93</td>
</tr>
</tbody>
</table>

Table 7.3: Normal leave, normal join, and migration times (sec).

The delay of a normal leave or join event is calculated by subtracting the execution time of a non-adaptive execution starting at the adaptation point from the execution time of the adaptive execution starting at the same adaptation point. The main result is that the cost of an adaptation is typically on the order of 0.1 to 3 seconds for this set of applications.

The above measurements are performed with a sufficiently long grace period to ensure that all leaves are normal leaves. For the applications in Table 7.1, the average time between successive adaptation points is 0.1-0.2 seconds for MGS, Jacobi and 3D-FFT, 1.4 seconds for Water, 2.6 seconds for NBF, and about 4.4 seconds for Barnes.

We conclude that with reasonable values of the grace period, the system supports
Figure 7.3: Execution times for different intervals between adapt events.
rates of adapt events of several adaptations per minute without significant performance degradation. In addition, the cost of adaptation has to be weighed against the flexibility of using additional nodes as they become available, or the ability to continue when a node withdraws.

**Cost of migration**

The cost of adaptation by migration alone is substantially higher than that of adaptation by normal leaves and joins.

We measured the cost of migrating a thread from one node to another. Two components determine the direct cost of migration: (i) the cost to create a new thread on the new host (approximately 0.6 to 0.8 seconds), and (ii) the cost to move the thread image (at a rate of approx. 7.8 MByte/s). The results are reported in Table 7.3, and have to be compared against the results in the same table for normal leaves. We see that, with the exception of Barnes, where the costs are comparable, the cost of migration is substantially higher than that of a normal leave.

The total cost of an urgent leave is the sum of the migration cost above plus the cost of a normal leave (performed at the next adaptation point) plus the cost of multiplexing until the adaptation point is reached (see Figure 7.1.c).

The main benefit of adapting with normal leaves and joins is that the application can change the number of threads during its execution. That processing of the joins and normal leaves is a few seconds faster than the direct cost of migration is an additional advantage. Furthermore, if we only had migration, the execution time after the adaptation might well double after the adaptation: the multiplexing shown in Figure 7.1.c continues until the end or until another node becomes available.

**7.3.2 Micro analysis of adaptation costs**

To understand the cost of handling adapt events, we report some detailed execution statistics in Tables 7.4 through 7.7.
The advantage of a SDSM — automatic data movement — also makes it difficult to obtain accurate statistics for the real cost of adaptation (the sum of the cost of maintaining the consistency information and the transfer cost for all pages). Unfortunately, simply counting the number of page fetches in both adaptive and non-adaptive runs does not provide a good indication of the overhead. For instance, when a page fetch occurs after an adaptation, it is difficult to distinguish whether this page fetch would have occurred even without the adaptation, or whether it was caused by the adaptation. In fact, the garbage collection and the relocation of data on a leave may cause certain page fetches not to occur after adaptation, while they would have occurred in a non-adaptive execution. Furthermore, since we move several pages in a single message when handling a leave event, it is actually possible that a program runs faster with adaptation than without (see, e.g., Water in Table 7.4), if the leave event moved pages that would have been fetched in any case.

We therefore use the following method to measure the data movement associated with a single adaptation from $m$ to $n$ threads. We perform an adaptive run, and we start recording statistics at the adaptation point. Then, we perform a non-adaptive run with $n$ threads with statistics-recording started at the same point during the execution as where the adaptation occurred in the adaptive run. The difference between the statistics recorded in these two experiments reflects exactly the effects of the adaptation. At the time statistics recording starts, the application has performed exactly the same amount of work in both experiments, albeit with $m$ threads in the adaptive run and $n$ in the non-adaptive run. The statistics recorded therefore show exactly the same amount of remaining work performed on $n$ threads, plus the data movement as a result of adaptation in the case of the adaptive run.

For each of the applications, Tables 7.4 through 7.6 provide detailed statistics obtained using these experiments. In particular, Tables 7.4 and 7.5 provides statistics for one thread leaving, or two threads leaving at the same adaptation point, respectively, always starting with eight threads.
In each case, a maximum and a minimum case are reported, depending on the thread id of the leaving threads. The statistics reported are the difference in execution time, the number of pages moved when handling the leave event (Pages-System) and the extra number of pages that are fetched subsequently during execution of the application (Pages-Application). Table 7.6 shows statistics for one thread joining and two threads joining at the same adaptation point, in each case ending up with a total of eight threads after adaptation. We also show the case of one thread leaving and one thread joining at the same adaptation point. Finally, Table 7.7 shows for a single application, 3D-FFT, the same statistics but for differing numbers of threads before and after adaptation.

<table>
<thead>
<tr>
<th></th>
<th>Time (sec)</th>
<th>Minimum</th>
<th></th>
<th>Time (sec)</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pages</td>
<td>System</td>
<td>Pages</td>
<td>Pages</td>
<td>System</td>
</tr>
<tr>
<td></td>
<td>Appl.</td>
<td></td>
<td>Appl.</td>
<td>Appl.</td>
<td></td>
</tr>
<tr>
<td>MGS</td>
<td>1.62</td>
<td>576</td>
<td>3955</td>
<td>1.93</td>
<td>576</td>
</tr>
<tr>
<td>Jacobi</td>
<td>1.53</td>
<td>1526</td>
<td>3383</td>
<td>2.29</td>
<td>1526</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>0.93</td>
<td>672</td>
<td>2148</td>
<td>1.3</td>
<td>672</td>
</tr>
<tr>
<td>NBF</td>
<td>1.14</td>
<td>1552</td>
<td>3239</td>
<td>1.32</td>
<td>1552</td>
</tr>
<tr>
<td>Barnes</td>
<td>1.75</td>
<td>0</td>
<td>538</td>
<td>2.46</td>
<td>0</td>
</tr>
<tr>
<td>Water</td>
<td>-0.41</td>
<td>34</td>
<td>-28</td>
<td>0.08</td>
<td>35</td>
</tr>
</tbody>
</table>

Table 7.4: Costs for one leave (8 → 7).

<table>
<thead>
<tr>
<th></th>
<th>Time (sec)</th>
<th>Minimum</th>
<th></th>
<th>Time (sec)</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pages</td>
<td>System</td>
<td>Pages</td>
<td>Pages</td>
<td>System</td>
</tr>
<tr>
<td></td>
<td>Appl.</td>
<td></td>
<td>Appl.</td>
<td>Appl.</td>
<td></td>
</tr>
<tr>
<td>MGS</td>
<td>2.04</td>
<td>1152</td>
<td>3258</td>
<td>2.24</td>
<td>1152</td>
</tr>
<tr>
<td>Jacobi</td>
<td>1.89</td>
<td>3046</td>
<td>4071</td>
<td>3.22</td>
<td>3050</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>1.07</td>
<td>1408</td>
<td>2012</td>
<td>1.68</td>
<td>1408</td>
</tr>
<tr>
<td>NBF</td>
<td>0.96</td>
<td>3104</td>
<td>2229</td>
<td>2.23</td>
<td>3104</td>
</tr>
<tr>
<td>Barnes</td>
<td>1.23</td>
<td>0</td>
<td>890</td>
<td>1.44</td>
<td>0</td>
</tr>
<tr>
<td>Water</td>
<td>0.03</td>
<td>69</td>
<td>-20</td>
<td>0.21</td>
<td>72</td>
</tr>
</tbody>
</table>

Table 7.5: Costs for two leaves (8 → 6).
<table>
<thead>
<tr>
<th>Application</th>
<th>One thread joining</th>
<th>Two threads joining</th>
<th>One leave and one join</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec)</td>
<td>Pages Appl.</td>
<td>Time (sec)</td>
</tr>
<tr>
<td>MGS</td>
<td>2.30</td>
<td>4040</td>
<td>1.76</td>
</tr>
<tr>
<td>Jacobi</td>
<td>1.73</td>
<td>6101</td>
<td>2.2</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>1.16</td>
<td>3144</td>
<td>1.39</td>
</tr>
<tr>
<td>NBF</td>
<td>0.48</td>
<td>6327</td>
<td>1.16</td>
</tr>
<tr>
<td>Barnes</td>
<td>1.08</td>
<td>958</td>
<td>1.27</td>
</tr>
<tr>
<td>Water</td>
<td>0.01</td>
<td>-61</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Table 7.6: Costs for one join (7 → 8), two joins (6 → 8) and a leave/join (6 → 6).

<table>
<thead>
<tr>
<th># Threads before &amp; after adaptation</th>
<th>Time (sec)</th>
<th>Pages System</th>
<th>Pages Appl.</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 5</td>
<td>Min 1.28</td>
<td>1210</td>
<td>1776</td>
</tr>
<tr>
<td>6 5</td>
<td>Max 1.89</td>
<td>1210</td>
<td>3338</td>
</tr>
<tr>
<td>6 4</td>
<td>Min 1.33</td>
<td>2420</td>
<td>1755</td>
</tr>
<tr>
<td>6 4</td>
<td>Max 2.64</td>
<td>2420</td>
<td>4787</td>
</tr>
<tr>
<td>4 3</td>
<td>Min 2.03</td>
<td>1760</td>
<td>1868</td>
</tr>
<tr>
<td>4 3</td>
<td>Max 2.33</td>
<td>1760</td>
<td>2874</td>
</tr>
<tr>
<td>4 2</td>
<td>Min 2.51</td>
<td>3520</td>
<td>1304</td>
</tr>
<tr>
<td>4 2</td>
<td>Max 4.28</td>
<td>3520</td>
<td>3584</td>
</tr>
<tr>
<td>5 6</td>
<td>Min 1.30</td>
<td>n/a</td>
<td>3585</td>
</tr>
<tr>
<td>4 6</td>
<td>Max 1.62</td>
<td>n/a</td>
<td>5490</td>
</tr>
<tr>
<td>3 4</td>
<td>Min 2.12</td>
<td>n/a</td>
<td>3473</td>
</tr>
<tr>
<td>2 4</td>
<td>Max 2.95</td>
<td>n/a</td>
<td>5378</td>
</tr>
</tbody>
</table>

Table 7.7: 3D-FFT: Costs for leaves and joins.
Data transfer

The key component in handling an adaptation is additional network traffic. In almost all cases, the number of pages transferred by the application after the adaptation (labeled Pages-Application) dominates the number of pages transferred during the actual adaptation (labeled Pages-System). This observation suggests that better thread id re-assignment strategies, which reduce the amount of application data transfer, could have an important beneficial effect on adaptation times.

Size of the system

The cost of adaptation decreases as the number of threads increases. Table 7.7 shows the cost of handling leaves (top) and joins (bottom); the cost increases as the number of threads is reduced. For many applications, the total amount of data to be re-distributed for any adaptation from $x$ to $x + n$ threads tends to grow as $x$ decreases. Furthermore, fewer links are available to carry this traffic.

Batching joins and leaves

The cost per adaptation decreases as more threads join or leave at the same time. Looking at Tables 7.4, 7.5 and 7.6, we see that handling two events together is cheaper than handling two single events.* This observation applies when the two events are both leaves, or both joins, but it especially applies where there is a leave and a join, because of the optimization applied in this case (see Section 7.2.2).

As a result, since the numbers reported in Section 7.3.1 always report on a single leave or a single join at an adaptation point, they must be considered somewhat pessimistic.

---

*The exception is Water, where adaptations are very cheap to begin with. Also, such minor time differences may be due to the measurement setup.
Other factors

The cost of adaptation decreases as more adaptations happen during the execution. Many applications do not access all the shared memory they own in every iteration. At a leave, only the accessed pages need to be transferred, even if the leaving thread’s data partition is much larger. So the more recently a thread had previously joined, the fewer data must be transferred.

Furthermore, the thread id reassignment algorithm, the ids of leaving/joining threads, the number of pages owned by leaving threads, the number of pages accessed after adaptation — all affect the cost of adaptation.

7.4 Related work

Various systems have been developed to allow sequential computations to use idle time on networked nodes. These systems include, among others, Butler [Nic87], Condor [LLM88], and many thread migration systems such as, e.g. Sprite [DO87]. Our work distinguishes itself from these systems by its support for parallel computations.

Cilk-NOW [BL97], Dataparallel-C [NQ93], Piranha [CFGK95], and various migration-based systems (e.g., Millipede [ISS97] or versions of PVM [KOW97]) support adaptive parallel computation on NOWs.

Blumofe and Lischicki [BL97] describe the Cilk-NOW system for adaptive and reliable parallel execution of functional Cilk programs on networks of workstations. Programs need to be written in the Cilk language. Only functional Cilk programs are supported by Cilk-NOW. Nodes may join and leave an ongoing computation at any time, although, as with all systems, there is some “lag time” before the computation effectively leaves the node. Joining nodes “steal” work, under the form of a closure, from a randomly chosen node that is already participating in the computation. Leaving nodes return their unfinished closures to one of the remaining nodes. Various mechanisms make this work stealing efficient. Fault tolerance is achieved by a combination of transactional subcomputations and checkpointing. In contrast to Cilk-
NOW, our system is not restricted to functional programs. Furthermore, although unmodified Cilk programs can be run on Cilk-NOW, our system has the advantage of supporting unmodified programs written in an industry standard form, based on a more general programming model. Finally, we take advantage of the inevitable “lag time” to attempt a more efficient form of adaptation.

Nedeljkovic and Quinn [NQ93] describe a system for executing Dataparallel C programs on NOWs [HQ91]. Dataparallel C contains data distribution statements, and programs are compiled to execute in parallel on virtual processors. Each virtual processor executes a C program augmented with communication statements inserted by the compiler. The runtime system then allocates some number of virtual processors to physical processors. To adapt to various load conditions, the number of virtual processors on a particular physical processor is adjusted. In addition, the data sections associated with a moving virtual processor must be moved as well. In the absence of an underlying shared memory platform, Dataparallel C is limited to programs for which the compiler can fully analyze the program to the point where it can generate communication statements, and discover exactly what data must be moved when a virtual processor moves. Furthermore, the use of virtual processors deprives the compiler of optimization opportunities. Our system avoids this limitation by using an underlying shared memory platform.

Piranha [CFGK95] supports adaptive parallelism for programs using the Linda tuple space. In Piranha, the programmer needs to provide three routines: a feeder routine to oversee the adaptive computation, a piranha routine to perform the actual computation, and a retreat routine to output to the tuple space whatever information is necessary for the computation to continue after a thread leaves. Piranha threads may come and go at any time. Piranha requires the adoption of the tuple space as a parallel programming model, and, in addition, requires the programmer to write special code to achieve adaptivity (the retreat routine, and some modifications to the piranha routine for mutual exclusion). Instead, our system uses an industry
standard programming model, and requires no modifications.

As compared to migration-based systems [ISS97, KOW97], we use migration only if the grace period expires. We have documented the benefits of doing so, compared to the relatively small cost.

Fully automatic data management distinguishes our approach from systems such as Adaptive Multiblock PARTI [EAS+97], where the application programmer must add communication schedules by hand. Also, this system requires a skeleton thread to be left on a leaving node where our system can completely remove threads from a node.

Ioannidis and Dwarkadas use a SDSM as a platform for load balancing [ID98]. To adjust the load (e.g., in response to competing use on a node), the iterations of a loop are partitioned based on a sophisticated strategy that tries to avoid rebalancing the computation too often. Their system explicitly deals with competing loads on a node, but does not handle the departure of a node.

7.5 Summary

Our system provides transparent adaptive parallel execution of OpenMP programs on NOWs. This system is a version of the TreadMarks SDSM augmented to support adaptivity. The system works through iteration re-partitioning at the end of parallel loops — its efficient mode of operation — or through migration — a less efficient procedure executed only when necessary.

It is instructive to observe what properties of the resulting system are dependent on which design decisions. The use of OpenMP, besides being an industry-standard, allows transparent iteration re-partitioning, because the compiler generates the iteration-partitioning code such that it is executed at the beginning of each parallel construct. It is possible for the user to write TreadMarks code that does the same, but this is not the typical way TreadMarks programs are written (iteration partitioning is done once in many programs, as soon as the number of nodes becomes
known). This result is not specific to OpenMP: the same techniques can be used for other contexts that do not fix the number of threads (nodes), e.g., when compiling the Fortran90 or HPF array statement [WSC96].

The use of TreadMarks allows automatic distribution and communication of data, both during regular computation and after adaptation. Otherwise, the compiler would face the difficult task of generating communication code, or the user would have to write it.

The use of a grace period allows the system to most often execute adaptations by iteration re-partitioning, with migration as a backup solution.

We have described a prototype implementation and demonstrated that it exhibits good performance in a small NOW. There is no cost for the provision of adaptivity in the absence of adaptation. The performance penalty incurred for moderate rates of adaptations appears acceptable in the face of the augmented functionality.

Our current system delivers good performance but many opportunities for improvement are yet to be explored. Better thread id reassignment strategies offer much room for improved performance. The grace period also gives rise to a new use of compiler optimization that we have started to explore. In this paper, we equate adaptation points with the entry or exit into a concurrent construct, i.e., in many cases, the start or end of a parallel loop written by the user. However, the compiler can control the frequency of adaptation points by transformations similar to loop tiling or strip mining. Depending on the degree of flexibility required, the compiler can generate code that determines at runtime the trip counts or tiling of the loops, subject to the characteristics of the execution environment.
Chapter 8

Conclusions and Future Work

8.1 Conclusions

The OpenMP Application Programming Interface (API) is an emerging standard for parallel programming on shared memory architectures. Networks of workstations (NOWs) are attractive parallel programming platforms because of their good price/performance ratio, as well as their flexibility and their potential to scale. This work is the first to extend the support for OpenMP to networks of workstations. Our design is based on integrating the compiler and the run-time system. In the combined system, the run-time library remains the basic vehicle for implementing shared memory, while the compiler performs optimization rather than implementation.

The integrated compiler and run-time approach can effectively optimize irregular applications, for which an exact compile-time analysis is not possible. We have designed and implemented an optimization to aggregate inter-process communication into fewer message exchanges for irregular OpenMP programs. This approach is based on a modified software distributed shared memory layer, and fairly simple compile-time support. The only required compile-time support is regular section analysis of the indirection arrays. Run-time support for dynamic detection of changes to the indirection array, as well as to the shared data, eliminates any unnecessary computation and communication. We have also designed and implemented an optimization that improves the performance of OpenMP on large scale clusters of workstations. We do so by eliminating the contention caused by sequential sections of the program. Our solution replicates the sequential computation on all threads, so that everyone modifies their locally cached copy of the shared data and does not have to go to
the master thread for updates. We take advantage of the fact that all threads execute the same deterministic code to make efficient use of multicast. In addition, the fork-join execution model of OpenMP allows us to detect and replicate the sequential executions at runtime.

The integrated system also simplifies the run-time implementation. We have extended the base OpenMP system from networks of uniprocessors to networks of shared memory multiprocessors (SMPs). The compile-time information greatly reduces the number of changes required to the run-time system in order to exploit the intra-node hardware shared memory. We have also extended the base OpenMP implementation to allow programs to automatically adapt to a variable number of computing nodes during the execution. The implementation exploits the convenient adaptation points in OpenMP programs, points at which the cost of adaptation can be much reduced.

8.2 Future Work

Systems such as the DOE’s ASCI machines are constructed from hundreds to thousands shared-memory multiprocessors connected by high speed networks. MPI is currently the only standard parallel programming interface supported on these machines. Supporting OpenMP on such super-clusters will improve the productivity of developing codes in areas ranging from defense, drug design, genetic engineering, material science to financial modeling, for which many challenging computations cannot be performed on any other platform. Even for the performance-critical applications, OpenMP can be used for quick proof-of-concept parallelization.

Two important problems have to be solved in order to provide OpenMP support on super-clusters. First, fundamental changes to the run-time directory scheme must be made. Currently, the directory that keeps the consistency information is replicated on each node, making it impossible to handle tera-bytes of shared data. The solution is to distribute the directory information among the nodes. The design of the directory hierarchy and the directory caching policy must maximize the hit rate on locally
cached entries, and minimize the number of messages on a cache miss. Second, the system must have efficient support for parallel I/O, which is crucial for accessing large data files in grand challenge applications. Application-level information is necessary in order to optimize the partitioning of file data among nodes, to optimize the physical file layout on disks, and to aggregate the I/O of application data structures. An integrated compiler and runtime approaches can be used to predict access patterns and automatically enable these optimizations.
Bibliography


