TreadMarks: Shared Memory Computing on Networks of Workstations

Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher,
Honghui Lu, Ramakrishnan Rajamony, Weimin Yu and Willy Zwaenepoel
Rice University
treadmarks@ece.rice.edu
http://www.cs.rice.edu/~willy/TreadMarks/overview.html

Abstract

TreadMarks supports parallel computing on networks of workstations by providing the application with a shared memory abstraction. Shared memory facilitates the transition from sequential to parallel programs. After identifying possible sources of parallelism in the code, most of the data structures can be retained without change, and only synchronization needs to be added to achieve a correct shared memory parallel program. Additional transformations may be necessary to optimize performance, but this can be done in an incremental fashion. We discuss the techniques used in TreadMarks to provide efficient shared memory, and our experience with two large applications, mixed integer programming and genetic linkage analysis.

1 Introduction

High-speed networks and rapidly improving microprocessor performance make *networks of workstations* an increasingly appealing vehicle for parallel computing. By relying solely on commodity hardware and software, networks of workstations offer parallel processing at a relatively low cost. A network-of-workstations multiprocessor may be realized as a *processor bank*, a number of processors dedicated for the purpose of providing computing cycles. Alternatively, it may consist of a dynamically varying set of machines on which idle cycles are used to perform long-running computations. In the latter case, the (hardware) cost is essentially zero, since many organizations already have extensive workstation networks in place. In terms of performance, improvements in processor and network speed allow networked workstations to deliver performance approaching or exceeding supercomputer performance for an increasing class of applications. It is by no means our position that such loosely coupled multiprocessors will render obsolete more tightly coupled designs. In particular, the lower latencies and higher bandwidths of these tightly coupled designs allow efficient execution of applications with more stringent synchronization and communication

This research was supported in part by the National Science Foundation under Grants CCR-9116343, CCR-9211004, CDA-9222911, CDA-9222911, and CDA-9502791, and by the Texas Advanced Technology Program and Tech-Sym Inc. under Grant 003604012.

requirements. However, we argue that the advances in networking technology and processor performance will greatly expand the class of applications that can be executed efficiently on a network of workstations.

In this paper, we discuss our experience with parallel computing on networks of workstations using the TreadMarks *distributed shared memory* (DSM) system. DSM allows processes to assume a globally shared virtual memory even though they execute on nodes that do not physically share memory [9]. Figure 1 illustrates a DSM system consisting of N networked workstations, each with its own memory, connected by a network. The DSM software provides the abstraction of a globally shared memory, in which each processor can access any data item, without the programmer having to worry about where the data is, or how to obtain its value. In contrast, in the "native" programming model on networks of workstations, *message passing*, the programmer must decide *when* a processor needs to communicate, with *whom* to communicate, and *what* data to send. For programs with complex data structures and sophisticated parallelization strategies, this can become a daunting task. In a DSM system, the programmer can focus on algorithmic development rather than on managing partitioned data sets and communicating values. In addition to ease of use, DSM provides the same programming environment as that available on (hardware) shared-memory multiprocessors, allowing programs written for a DSM to be ported easily to a shared-memory multiprocessor. Porting a program from a hardware shared-memory multiprocessor to a DSM system may require more modifications to the program, because the much higher latencies in a DSM system put an even greater value on locality of memory access.

The programming interfaces to DSM systems may differ in a variety of respects. We focus here on memory *structure* and memory *consistency model*. An unstructured memory appears as a linear array of bytes, whereas in a structured memory processes access memory in terms of objects or tuples. The memory model refers to how updates to shared memory become visible to the processes in the system. The most intuitive model of shared memory is that a read should always return the last value written. Unfortunately, the notion of "the last value written" is not well defined in a distributed system. A more precise notion is *sequential consistency*, whereby the memory appears to all processes as if they were executing on a single multiprogrammed processor [7]. With sequential consistency, the notion of "the last value written" is precisely defined. The simplicity of this model may, however, exact a high price in terms of performance, and therefore much research has been done into *relaxed memory models*.

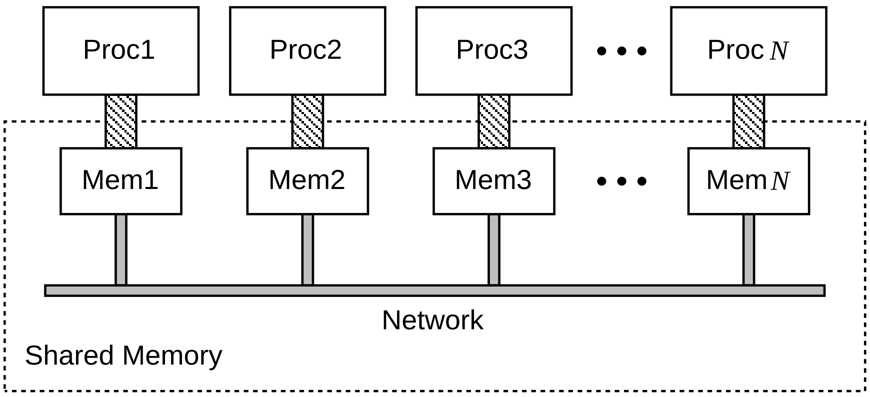

Figure 1 Distributed Shared Memory: Each processor sees a shared address space, denoted by the dashed outline, rather than a collection of distributed address spaces.

In terms of implementation techniques, one of the primary distinguishing features of a DSM system is whether or not it uses the virtual memory page protection hardware to detect accesses to shared memory. The naive use of virtual memory protection hardware may lead to poor performance because of discrepancies between the page size of the machine and the granularity of sharing in the application.

The system discussed in this paper, TreadMarks [5], provides shared memory as a linear array of bytes. The memory model is a relaxed memory model, namely *release consistency*. The implementation uses the virtual memory hardware to detect accesses, but it uses a *multiple writer* protocol to alleviate the problems resulting from mismatch between the page size and the application's granularity of sharing.

TreadMarks runs at user-level on Unix workstations. No kernel modifications or special privileges are required, and standard Unix interfaces, compilers, and linkers are used. As a result, the system is fairly portable. In particular, it has been ported to a number of platforms, including IBM RS-6000, IBM SP-1, IBM SP-2, DEC Alpha, DEC DECStation, HP, SGI, and SUN-Sparc.

This paper first describes the application programming interface provided by TreadMarks (Section 2). Next, we discuss some of the performance problems observed in DSM systems using conventional sequential consistency (Section 3) and the techniques used to address these problems in TreadMarks (Sections 4 and 5). We briefly describe the implementation of TreadMarks in Section 6, and some basic operation costs in the experimental environment used in Section 7. We demonstrate TreadMarks' efficiency by discussing our experience with two large applications, mixed integer programming and genetic linkage analysis (Section 8). Finally, we discuss related work in Section 9, and we offer some conclusions and directions for further work in Section 10.

2 Shared Memory Programming

2.1 Application Programming Interface

The TreadMarks API is simple but powerful (see Figure 2 for the C language interface). It provides facilities for process creation and destruction, synchronization, and shared memory allocation. Shared memory allocation is done through `Tmk_malloc()`. Only memory allocated by `Tmk_malloc()` is shared. Memory allocated statically or by a call to `malloc()` is private to each process.

We focus on the primitives for synchronization. Synchronization is a way for the programmer to express ordering constraints between the shared memory accesses of different processes. A simple form of synchronization occurs with *critical sections*. A critical section guarantees that only one process at a time executes inside the critical section. Such a construct is useful when multiple processes are updating a data structure, and concurrent access is not allowed.

More formally, two shared memory accesses are *conflicting* if they are issued by different processors to the same memory location and at least one of them is a write. A parallel program has a *data race* if there is no synchronization between two conflicting accesses. Data races are often bugs in the program, because the final outcome of the execution is timing-dependent. If, for example, one of the accesses is a read and the other is a write and no synchronization is present between the read and the write, it may be that either the read or the write execute first, with different outcomes from each execution. Data races can be avoided by introducing synchronization.

TreadMarks provides two synchronization primitives: barriers and exclusive locks. A process waits at a barrier by calling `Tmk_barrier()`. Barriers are global: the calling process is stalled until all processes in the system have arrived at the same barrier. A `Tmk_lock_acquire` call acquires a lock for the calling process, and `Tmk_lock_release` releases it. No process can acquire a lock while another process is holding it. A lock can be used to implement critical sections, as described above.

```
/* the maximum number of parallel processes supported by TreadMarks */

#define TMK_NPROCS

/* the actual number of parallel processes in a particular execution */

extern  unsigned        Tmk_nprocs;

/* the process id, an integer in the range 0 ... Tmk_nprocs - 1 */

extern  unsigned        Tmk_proc_id;

/* the number of lock synchronization objects provided by TreadMarks */

#define TMK_NLOCKS

/* the number of barrier synchronization objects provided by TreadMarks */

#define TMK_NBARRIERS

/* Initialize TreadMarks and start the remote processes */

void    Tmk_startup(int argc, char **argv)

/* Terminate the calling process.  Other processes are unaffected. */

void    Tmk_exit(int status)

/* Block the calling process until every other process arrives at the barrier. */

void    Tmk_barrier(unsigned id)

/* Block the calling process until it acquires the specified lock. */

void    Tmk_lock_acquire(unsigned id)

/* Release the specified lock. */

void    Tmk_lock_release(unsigned id)

/* Allocate the specified number of bytes of shared memory */

char   *Tmk_malloc(unsigned size)

/* Free shared memory allocated by Tmk_malloc. */

void    Tmk_free(char *ptr)
```

Figure 2 TreadMarks C Interface

This particular choice of synchronization primitives is not in any way fundamental to the design of TreadMarks; other primitives may be added later. We demonstrate the use of these and other TreadMarks primitives with two simple applications.

2.2 Two Simple Illustrations

Figures 3 and 4 illustrate the use of the TreadMarks API for Jacobi iteration and for solving the traveling salesman problem (TSP). Jacobi illustrates the use of barriers, while TSP provides an example of the use of locks. We are well aware of the overly simplistic nature of these example codes. They are included here for demonstration purposes only. Larger applications are discussed in Section 8.

Jacobi is a method for solving partial differential equations. Our example iterates over a two-dimensional array. During each iteration, every matrix element is updated to the average of its nearest neighbors (above, below, left, and right). Jacobi uses a scratch array to store the new values computed during each iteration, so as to avoid overwriting the old value of the element before it is used by its neighbor. In the parallel version, all processors are assigned roughly equal size bands of rows. The rows on the boundary of a band are shared by two neighboring processes.

The TreadMarks version in Figure 3 uses two arrays: a `grid` and a `scratch` array. `grid` is allocated in shared memory, while `scratch` is private to each process. `grid` is allocated and initialized by process 0. Synchronization in Jacobi is done by means of barriers. `Tmk_barrier(0)` guarantees that the initialization done by process 0 is completed before processes start computing. `Tmk_barrier(1)` makes sure that no processor overwrites any value in `grid` before all processors have read the value computed in the previous iteration. In the terminology introduced in Section 2.1, it avoids a data race between the reads of the `grid` array in the first nested loop and the writes in the second nested loop. `Tmk_barrier(2)` prevents any processor from starting the next iteration before all of the `grid` values computed in the current iteration are written. In other words, it avoids a data race between the writes of `grid` in the second nested loop and the reads of `grid` in the first nested loop of the next iteration.

The Traveling Salesman Problem (TSP) finds the shortest tour that starts at a designated city, passes through every other city on the map exactly once and returns to the original city. A simple branch and bound algorithm is used. The program maintains the length of the shortest tour found so far in `Shortest_length`. Partial tours are expanded one city at a time. If the current length of a partial tour plus a lower bound on the remaining portion of the tour is longer than the current shortest tour, the partial tour is not explored further, because it cannot possibly lead to a tour shorter than the current minimum length tour. The lower bound is computed by a fast conservative approximation of the length of the minimum spanning tree connecting all of the nodes not yet in the tour with themselves and with the first and last node in the current tour.

The sequential TSP program keeps a queue of partial tours, with the most promising one at the head. Promise is determined by the sum of the length of the current tour and the lower bound on the length to connect the remaining cities. The program keeps adding partial tours to the queue until a partial tour which is longer than a threshold number of cities is found at the head of the queue. It removes this partial tour from the queue and tries all permutations of the remaining cities. Next, it compares the shortest tour including this partial tour with the current shortest tour, and updates the current shortest tour if necessary. Finally, the program goes back to the tour queue and tries again to remove a promising partial tour of sufficient length from the queue.

Figure 4 shows pseudo code for the parallel TreadMarks TSP program. The shared data structures, the queue and the minimum length, are allocated by process 0. Exclusive access to these shared data structures is achieved by surrounding all accesses to them by a lock acquire and a lock

```
#define   M     1024
#define   N     1024
float     **grid;                   /* shared array */
float     scratch[M][N];            /* private array */

main()

{
    Tmk_startup();

    if( Tmk_proc_id == 0) {
        grid = Tmk_malloc( M*N*sizeof(float) );
        initialize grid;
    }

    Tmk_barrier(0);

    length = M / Tmk_nprocs;
    begin = length * Tmk_proc_id;
    end = length * (Tmk_proc_id+1);

    for( number of iterations ) {

        for( i=begin; i<end; i++ )
            for( j=0; j<N; j++ )
                scratch[i][j] = (grid[i-1][j]+grid[i+1][j]+
                                 grid[i][j-1]grid[i][j+1])/4;

        Tmk_barrier(1);

        for( i=begin; i<end; i++ )
            for( j=0; j<N; j++ )
                grid[i][j] = scratch[i][j];

        Tmk_barrier(2);

    }
}
```

Figure 3 Pseudo Code for the TreadMarks Jacobi Program release. All processes wait at `Tmk_barrier(0)` to make sure that these shared data structures are properly initialized before computation starts. Each process then acquires the queue lock to find a promising partial tour that is long enough so that it can be expanded sequentially. When such a tour is found, the process releases the queue lock. After expanding the current partial tour, a process acquires the lock on the minimum length, updates the minimum length if necessary, and then releases the lock. Finally, a process starts another iteration of the loop by acquiring the queue lock and finding another promising tour, and so forth until the queue is empty.

3 Implementation Challenges

DSM systems can migrate or replicate data to provide the abstraction of shared memory. Most DSM systems choose to replicate data, because this approach gives the best performance for a wide range of application parameters of interest [11]. With replicated data, the provision of memory *consistency* is at the heart of a DSM system: the DSM software must control replication in a manner that provides the abstraction of a *single* shared memory.

The *consistency model* defines how the programmer can expect the memory system to behave. The first DSM system, IVY [9], implemented *sequential consistency* [7]. In this memory model, processes observe shared memory *as if* they were executing on a multiprogrammed uniprocessor (with a single memory). In other words, there is a total order on all memory accesses, and that total order is compatible with the program order of memory accesses in each individual process.

In IVY's implementation of sequential consistency, the virtual memory hardware is used to maintain memory consistency. The local (physical) memories of each processor form a cache of the global virtual address space (see Figure 5). When a page is not present in the local memory of a processor, a page fault occurs. The DSM software brings an up-to-date copy of that page from its remote location into local memory and restarts the process. For example, Figure 5 shows the activity occurring as a result of a page fault at processor 1, which results in a copy of the necessary page being retrieved from the local memory of processor 3. IVY furthermore distinguishes read faults from write faults. With read faults, the page is replicated with read-only access for all replicas. With write faults, an invalidate message is sent to all processors with copies of the page. Each processor receiving this message invalidates its copy of the page and sends an acknowledgement message to the writer. As a result, the writer's copy of the page becomes the sole copy.

Because of its simplicity and intuitive appeal, sequential consistency is generally viewed as a "natural" consistency model. However, its implementation can cause a large amount of communication to occur. Communication is very expensive on a workstation network. Sending a message may involve traps into the operating system kernel, interrupts, context switches, and the execution of possibly several layers of networking software. Therefore, the number of messages and the amount of data exchanged must be kept low. We illustrate some of the communication problems in IVY using the examples from Section 2.2.

Consider for example the updates to the length of the current shortest tour in the TSP program in Figure 4. In IVY, this shared memory update causes invalidations to be sent to all other processors that cache the page containing this variable. However, since this variable is accessed only within the critical section protected by the corresponding lock, it suffices to send invalidations only to the next processor acquiring the lock, and only at the time of the lock acquisition.

The second problem relates to the potential for *false sharing*. False sharing occurs when two or more unrelated data objects are located in the same page and are written concurrently by separate processors. Since virtual memory pages are large, false sharing is a potentially serious problem. The Jacobi program in Figure 3 suffers from false sharing, if the `grid` array is laid out in memory

```
queue_type *Queue;
int        *Shortest_length;
int        queue_lock_id, min_lock_id;

main()
{
     Tmk_startup();
     queue_lock_id = 0;
     min_lock_id = 1;
     if (Tmk_proc_id == 0) {
         Queue = Tmk_malloc( sizeof(queue_type) );
         Shortest_length = Tmk_malloc( sizeof(int) );
         initialize Heap and Shortest_length;
     }
     Tmk_barrier(0);

     while( true ) do {
         Tmk_lock_acquire(queue_lock_id);
             if( queue is empty ) {
                 Tmk_lock_release(queue_lock_id);
                 Tmk_exit();
             }
             Keep adding to queue until a long,
             promising tour appears at the head;
             Path = Delete the tour from the head;
         Tmk_lock_release(queue_lock_id);

         length = recursively try all cities not on Path,
                  find the shortest tour length

         Tmk_lock_acquire(min_lock_id);
             if (length < *Shortest_length)
                 *Shortest_length = length;
         Tmk_lock_release(min_lock_id);
     }
}
```

Figure 4 Pseudo Code for the TreadMarks TSP Program

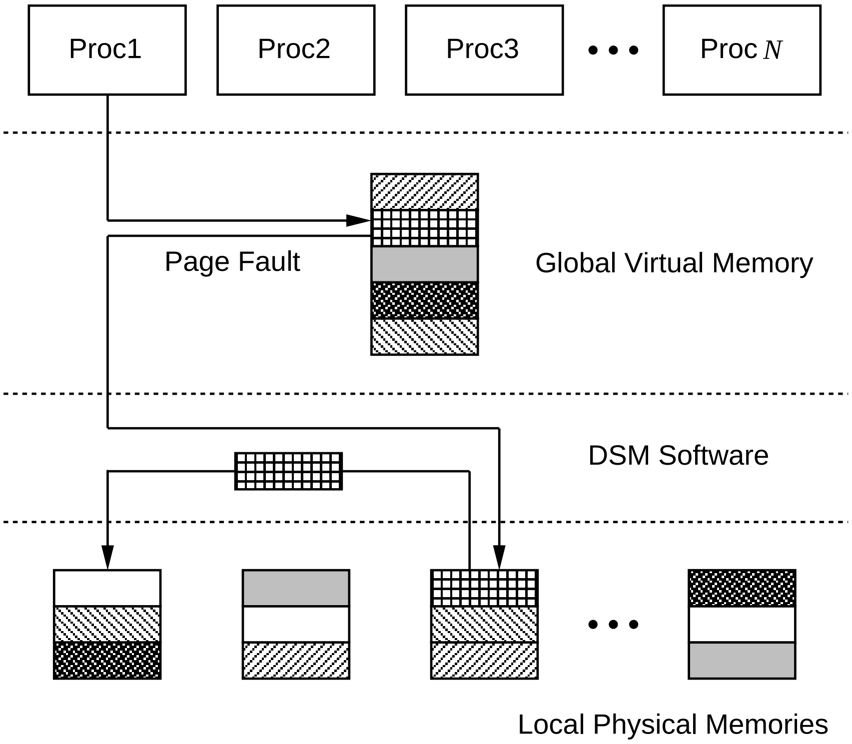

Figure 5 Operation of the IVY DSM System

such that portions of the array allocated to different processors lie within the same page. As both processors update their portion of the `grid` array, they are writing concurrently to the same page. Assume that initially processor P_1 holds the sole writable copy of the page. When processor P_2 writes to the page, it sends an invalidate message to processor P_1. P_1 sends the page to P_2 and invalidates its own copy. When P_1 writes next to the page, the same sequence of events will occur, with P_1 and P_2 interchanged. As each process writes to the page while it is held by the other process, the page will travel across the network. This repeated back and forth of the page is often referred to as the "ping-pong effect".

In order to address these problems, we have experimented with novel implementations of relaxed memory consistency models, and we have designed protocols to combat the false sharing problem. These approaches are discussed next.

4 Lazy Release Consistency

4.1 Release Consistency Model

The intuition underlying release consistency is as follows. Parallel programs should not have data races, because they may lead to wrong results. Thus, sufficient synchronization must be present to prevent data races. More specifically, synchronization must be present between two conflicting accesses to shared memory. Since this synchronization is present, there is no need to make any shared memory updates from one process visible to another process before they synchronize with each other, because the second process will not access the data until the synchronization operation has been executed.

We will illustrate this principle with the Jacobi and TSP examples from Section 2. Writes to shared memory in Jacobi occur after barrier 1 is passed, when the newly computed values are copied from the `scratch` array to the `grid` array (see Figure 3). This phase of the computation terminates when barrier 2 is passed. Barrier 2 is present to prevent processes from starting the next iteration before all the new values are written to the `grid` array. This barrier needs to be there for correctness (to avoid data races), regardless of the memory model. However, its presence allows us to delay notifying a process about updates to `grid` by another process until the barrier is lowered.

In TSP, the tour queue is the primary shared data structure. Processors fetch tasks from the queue and work on them, creating new tasks in the process. These newly created tasks are inserted into the queue. Updates to the task queue structure require a whole series of shared memory writes, such as its size, the task at the head of the queue, etc. Atomic access to the task queue data structure is required in order for correct program execution. Only one processor is permitted to access the task queue data structure at a time. This guarantee is achieved by putting a lock acquire and a lock release around these operations. In order to access the tour queue, a process needs to acquire the lock. It therefore suffices to inform the next process that acquires the lock of the changes to the tour queue, and this can be done at the time the lock is acquired.

These two examples illustrate the general principle underlying release consistency. Synchronization is introduced in a shared memory parallel program to prevent processes from accessing certain memory locations until the synchronization operation completes. From that it follows that it is not necessary to inform a process of modifications to those shared memory locations until the synchronization operation completes. If the program does not have data races, then it will appear as if the program executes on a sequentially consistent memory, the intuitive memory model that programmers expect. The above is true *on one condition:* all synchronization must be done using the TreadMarks supplied primitives. Otherwise, TreadMarks cannot tell when to make shared memory consistent.

4.2 Release Consistency Implementations

The definition of release consistency specifies the latest possible time when a shared memory update must become visible to a particular processor. This allows an implementation of release consistency considerable latitude in deciding when and how exactly a shared memory update gets propagated. TreadMarks uses the lazy release consistency algorithm [5] to implement release consistency. Roughly speaking, lazy release consistency enforces consistency at the time of an *acquire*, in contrast to the earlier implementation of release consistency in Munin [3], sometimes referred to as eager release consistency, which enforced consistency at the time of a release. Figure 6 illustrates the intuitive argument behind lazy release consistency. Assume that x is replicated at all processors. With eager release consistency a message needs to be sent to all processors at a release informing them of the change to x. However, only the next processor that acquires the lock can access x. With lazy release consistency, only that processor is informed of the change to x. In addition to the reduction in message traffic resulting from not having to inform all processes that cache a copy of x, lazy release consistency also allows the notification of modification to be piggybacked on the lock grant message going from the releasing to the acquiring process.

TreadMarks uses an *invalidate* protocol. At the time of an acquire, the modified pages are invalidated. A later access to that page causes an access miss which in turn causes an up-to-date copy of the page to be installed. An alternative would be to use an *update* protocol, in which the acquire message contains the new values of the modified pages. A detailed discussion of the

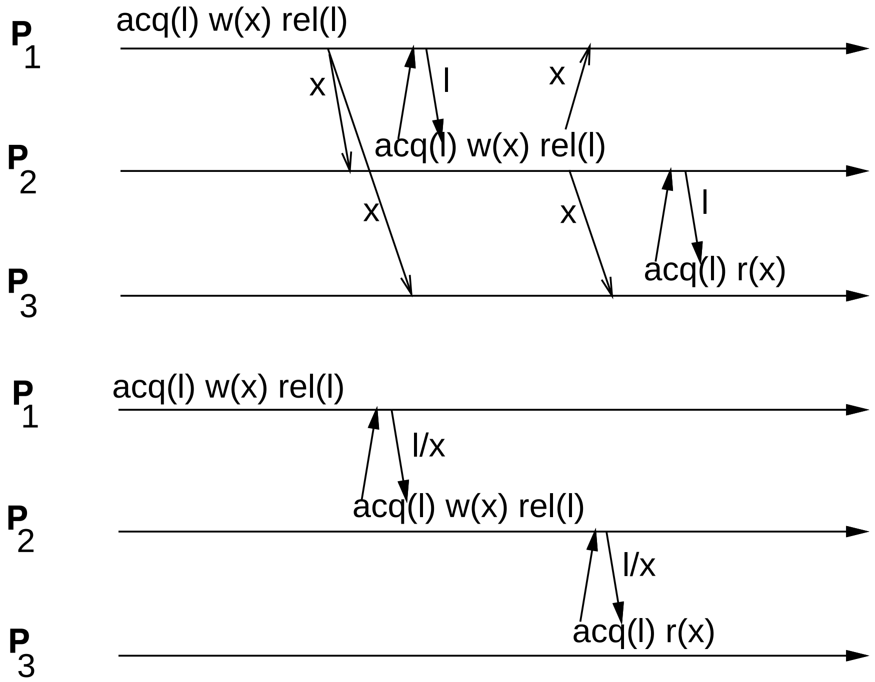

Figure 6 Eager vs. Lazy Release Consistency: The top portion of the figure illustrates eager release consistency, while the bottom portion illustrates lazy release consistency. The figure depicts the execution of three processes, P_1, P_2, and P_3, with the time axis going from left to right. The processes acquire and release the lock l, and read and write the variable x.

protocols used in TreadMarks is beyond the scope of this paper. We refer the reader to Keleher's thesis [5] for more detail. We will compare the performance of various implementations of release consistency with each other and with sequential consistency in Section 9.

5 Multiple-Writer Protocols

Most hardware cache and DSM systems such as IVY use *single-writer* protocols. These protocols allow multiple readers to access a given page simultaneously, but a writer is required to have sole access to a page before performing any modifications. Single-writer protocols are easy to implement because all copies of a given page are always identical, and page faults can always be satisfied by retrieving a copy of the page from any other processor that currently has a valid copy. Unfortunately, this simplicity often comes at the expense of message traffic. Before a page can be written, all other copies must be invalidated. These invalidations can then cause subsequent access misses if the processors whose pages have been invalidated are still accessing the page's data. *False sharing* can cause single-writer protocols to perform even worse because of interference between unrelated accesses. DSM systems typically suffer much more from false sharing than do hardware systems because they track data accesses at the granularity of virtual memory pages instead of cache lines.

As the name implies, *multiple-writer* protocols allow multiple processes to have, at the same time, a writable copy of a page. Assume that process P_1 and P_2 concurrently write to different locations within the same page, and assume furthermore that both P_1 and P_2 initially have an identical valid copy of the page. TreadMarks uses the virtual memory hardware to detect accesses and modifications to shared memory pages (see Figure 7). The shared page is initially write-protected. When a write occurs by P_1, TreadMarks creates a copy of the page, or a *twin*, and saves it as part of the TreadMarks data structures on P_1. It then unprotects the page in the user's address space, so that further writes to the page can occur without software intervention. When P_1 arrives at the barrier, we now have the modified copy in the user's address space and the unmodified twin. By doing a word-by-word comparison of the user copy and the twin, we can create a *diff*, a runlength encoding of the modifications to the page. Once the diff has been created, the twin is discarded. The same sequence of events happens on P_2. It is important to note that this entire sequence of events is local to each of the processors, and does not require any message exchanges, unlike in the case of a single-writer protocol. When P_1 and P_2 synchronize, for instance through a barrier, P_1 is informed that P_2 has modified the page, and vice versa, and they both invalidate their copy of the page. When they later access the page, they both take an access fault. The TreadMarks software on P_1 knows that P_2 has modified the page, sends a message to P_2 requesting the diff, and applies that diff to the page when it arrives. Again, the same sequence of events happens on P_2, with P_1 replaced by P_2 and vice versa. With the exception of the first time a processor accesses a page, its copy of the page is updated exclusively by applying diffs; a new complete copy of the page is never needed. The primary benefit of using diffs is that they can be used to implement multiple-writer protocols, thereby reducing the effects of false sharing. In addition, they significantly reduce overall bandwidth requirements because diffs are typically much smaller than a page.

The reader may wonder what happens when two processes modify overlapping portions of a page. We note that this corresponds to a data race, because it means that two processes write to the same location without intervening synchronization. Therefore, it is almost certainly an error in the program. Even on a sequentially consistent memory the outcome would be timing-dependent. The same is true in TreadMarks. It would be possible to modify TreadMarks such that it checks for such occurrences. If a diff arrives for a page that is locally modified, TreadMarks could check for overlap between the modifications, but this is currently not done.

6 The TreadMarks System

TreadMarks is implemented entirely as a user-level library on top of Unix. Modifications to the Unix kernel are not necessary because modern Unix implementations provide all of the communication and memory management functions required to implement TreadMarks at the user-level. Programs written in C, C++, or FORTRAN are compiled and linked with the TreadMarks library using any standard compiler for that language. As a result, the system is relatively portable. Currently, it runs on SPARC, DECStation, DEC/Alpha, IBM RS-6000, IBM SP-1, IBM SP-2, HP, and SGI platforms and on Ethernet, ATM, FDDI, and HIPPI networks. In this section, we briefly describe how communication and memory management are implemented by TreadMarks. For a more detailed discussion of the implementation, we refer the reader to Keleher's Ph.D. thesis [5].

By default, TreadMarks implements intermachine communication using UDP/IP through the Berkeley sockets interface. Since UDP/IP does not guarantee reliable delivery, TreadMarks uses light-weight, operation-specific, user-level protocols to insure message arrival. Every message sent by TreadMarks is either a request message or a response message. Request messages are sent by TreadMarks as a result of an explicit call to a TreadMarks library routine or a page fault. Once a machine has sent a request message, it blocks until a request message or the expected

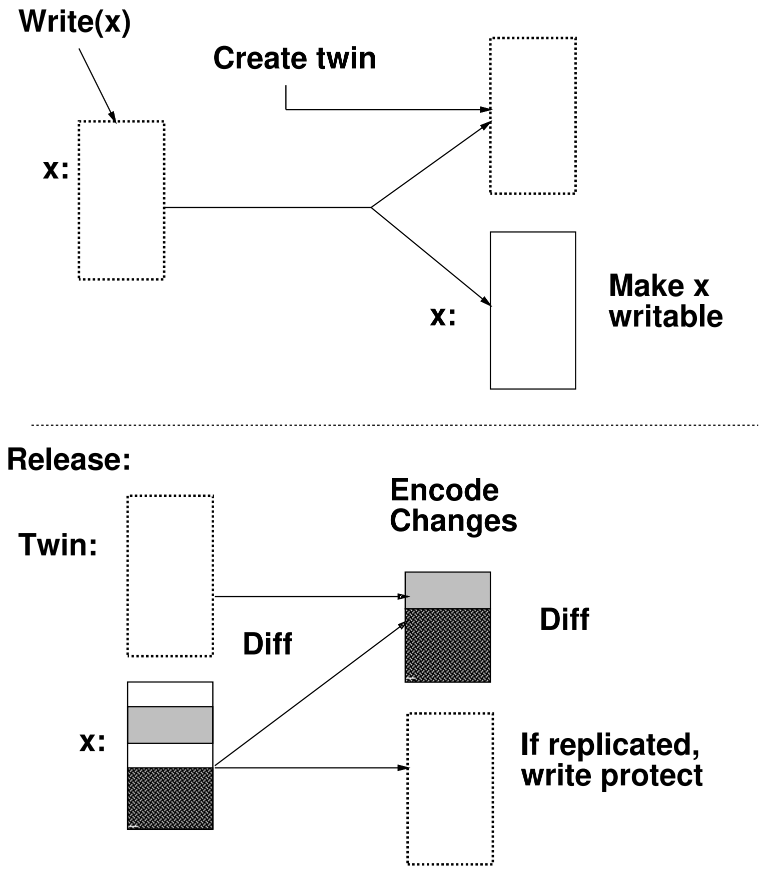

Figure 7 DiffCreation

response message arrives. If no response arrives within a certain timeout, the original request is retransmitted. To minimize latency in handling incoming requests, TreadMarks uses a `SIGIO` signal handler. Message arrival at any socket used to receive request messages generates a `SIGIO` signal. After the handler receives the request message, it performs the specified operation, sends the response message, and returns to the interrupted process.

To implement the consistency protocol, TreadMarks uses the `mprotect` system call to control access to shared pages. Any attempt to perform a restricted access on a shared page generates a `SIGSEGV` signal. The `SIGSEGV` signal handler examines local data structures to determine the page's state, and examines the exception stack to determine whether the reference is a read or a write. If the local page is invalid, the handler executes a procedure that obtains the necessary diffs from the minimal set of remote machines. If the reference is a read, the page protection is set to read-only. For a write, the handler allocates a page from the pool of free pages and performs a `bcopy` to create a *twin*. The same action is taken in response to a fault resulting from a write to a page in read-only mode. Finally, the handler upgrades the access rights to the original page and returns.

7 Basic Operation Costs

Our experimental environment consists of 8 DECstation-5000/240's, with a 4 kilobyte page size, running Ultrix V4.3. Each machine has a Fore ATM interface connected to a Fore ATM switch. The connection between the interface boards and the switch operates at 100 Mbps; the switch has an aggregate throughput of 1.2 Gbps. The interface board does programmed I/O into transmit and receive FIFOs, and requires messages to be assembled and disassembled from ATM cells by software. Interrupts are raised at the end of a message or a (nearly) full receive FIFO. Unless otherwise noted, the performance numbers describe 8-processor executions on the ATM LAN using the low-level adaptation layer protocol AAL3/4.

The minimum round-trip time using send and receive for the smallest possible message is 500 μseconds. Using a signal handler to receive the message at both processors, the round-trip time increases to 670 μseconds.

The minimum time to remotely acquire a free lock is 827 μseconds. The minimum time to perform an 8-processor barrier is 2186 μseconds. A remote access miss, to obtain a full page from another processor, takes 2792 μseconds.

The time to make a twin is 167 microseconds. The time to make a diff is somewhat data-dependent. If the page is unchanged, it takes 430 microseconds. If the entire page is changed, it takes 472 microseconds. The worst case occurs when every other word in the page is changed. In that case, making a diff takes 686 microseconds.

8 Applications

A number of applications have been implemented using TreadMarks, and the performance of some benchmarks has been reported earlier [5]. Here we describe our experience with two large applications that were recently implemented using TreadMarks. These applications, mixed integer programming and genetic linkage analysis, were parallelized, starting from an existing efficient sequential code, by the authors of the sequential code with some help from the authors of this paper. While it is difficult to quantify the effort involved, the amount of modification to arrive at an efficient parallel code proved to be relatively minor, as will be demonstrated in the rest of this section.

8.1 Mixed Integer Programming

Mixed integer programming (MIP) is a version of linear programming (LP). In LP, an objective function is optimized in a region described by a set of linear inequalities. In MIP, some or all of the variables are constrained to take on only integer values (sometimes just the values 0 or 1). Figure 8 shows a precise mathematical formulation.

The TreadMarks code to solve the MIP problem uses a *branch-and-cut* approach. The MIP problem is first relaxed to the corresponding LP problem. The solution of this LP problem will in general produce non-integer values for some of the variables constrained to be integers. The next step is to pick one of these variables, and branch off two new MIP problems, one with the added constraint that $x_i \leq \lfloor x_i \rfloor$ and another with the added constraint that $x_i \geq \lceil x_i \rceil$. Over time, the algorithm generates a tree of such branches. As soon as the algorithm finds a solution to the LP problem that satisfies the integer constraints, this solution establishes a *bound* on the final value of the objective function. Nodes in the branch tree for which the solution of the LP problem generates a result that is inferior to this bound need not be explored any further. In order to expedite this process, the algorithm uses a technique called *plunging*, essentially a depth-first search down the tree to find an integer solution and establish a bound as quickly as possible. One final algorithmic improvement of interest is the use of *cutting planes*. These are additional constraints added to the LP problem to tighten the description of the integer problem. The shared data structures are the current best solution and the queue of LP problems to be solved. Each is protected by a separate lock.

The code was used to solve all 51 of the problems from the MIPLIB library. This library includes representative examples from airline crew scheduling, network flow, plant location, fleet scheduling, etc. Figure 9 shows the speedups obtained for those problems in MIPLIB whose sequential running times are over 2,000 seconds. For most problems, the speedup is near-linear. One problem exhibits super-linear speedup, because the parallel code happens to hit on a solution early on in its execution, thereby pruning most of the branch-and-bound tree. For another problem, there is very little speedup, because the solution is found shortly after the pre-processing step, which is not (yet) parallelized. In addition to the problems from the MIPLIB library, the code was also used to solve a previously unsolved multicommodity flow problem. The problem took roughly 28 CPU days on an 8-processor IBM SP-1 and also exhibited near-linear speedup.

8.2 Genetic Linkage Analysis

Genetic linkage analysis is a statistical technique that uses family pedigree information to map human genes and locate disease genes in the human genome. Recent advances in biology and genetics have made an enormous amount of genetic material available, making computation the bottleneck in further discovery of disease genes.

In the classical Mendelian theory of inheritance, the child's chromosomes receive one strand of each of the parent's chromosomes. In reality, inheritance is more complicated due to *recombination*. When recombination occurs, the child's chromosome strand contains a piece of both of the

$$\begin{aligned}&\text{Minimize } c^T\, x \;+\; d^T\, y,\\ &\text{subject to } A\, x \;+\; B\, y \;\leq\; b,\\ &\text{where } x \;\in\; Z^p \text{ and } y \;\in\; R^q \text{ (sometimes } x \;\in\; \{0,1\}^p).\end{aligned}$$

Figure 8 The Mixed Integer Programming (MIP) Problem

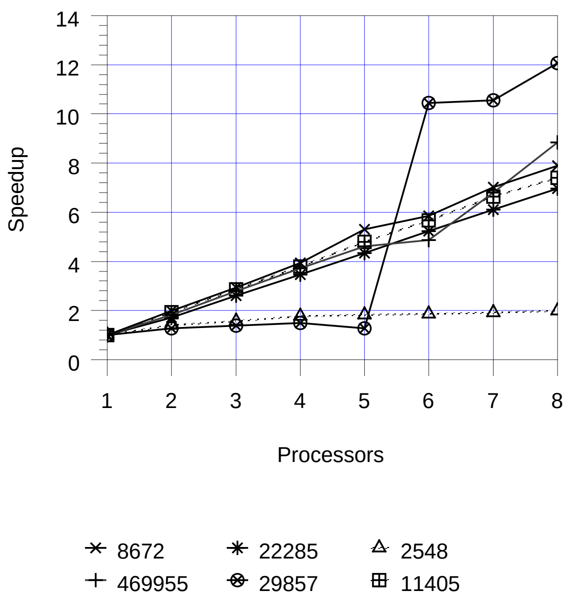

Figure 9 Speedup Results from the MIPLIB Library: Each line represents a different data set. The numbers at the bottom indicate the sequential execution in seconds for the corresponding data set. Only data sets with sequential running times larger than 2,000 seconds are presented.

strands of the parent's chromosome. The goal of linkage analysis is to derive the probabilities that recombination has occurred between the gene we are looking for and genes with known locations. From these probabilities an approximate location of the gene on the chromosome can be computed.

ILINK is a parallelized version of a widely used genetic linkage analysis program, which is part of the FASTLINK package [4]. ILINK takes as input a family tree, called a pedigree, augmented with some genetic information about the members of the family. It computes a maximum-likelihood estimate of θ, the recombination probability. At the top level, ILINK consists of a loop that optimizes θ. In each iteration of the optimization loop, the program traverses the entire pedigree, one nuclear family at a time, computing the likelihood of the current θ given the genetic information known about the family members. For each member of a nuclear family, the algorithm updates a large array of conditional probabilities, each representing the probability that the individual has certain genetic characteristics, conditioned on θ and on the part of the family tree already traversed.

The above algorithm is parallelized by splitting up the iteration space per nuclear family among the available processors in a manner that balances the load. Load balancing is essential and relies on knowledge of the genetic information represented in the array elements. An alternative approach, splitting up the tree traversal, failed to produce good speedups because most of the computation occurs in a small part of the tree (typically, the nodes closest to the root, representing deceased individuals for whom little genetic information is known).

Figure 10 presents speedups obtained for various data sets using ILINK. The data sets originate from actual disease gene location studies. For the data sets with a long running time, good speedups are achieved. For the smallest data sets, speedup is lower because the communication-to-computation ratio becomes larger. In general, the speedup is highly dependent on the communication-to-computation ratio, in particular on the number of messages per second. For the data set with the smallest speedup, ILINK exchanged approximately 1,800 messages per second, while for the data set with the best speedup the number of messages per second went down to approximately 300.

We found that the "overhead", i.e., time spent not executing application code, is dominated by idle time and Unix overhead. Idle time results from load imbalance and from waiting for messages to arrive over the network. Unix overhead is time spent in executing Unix library code and system calls. Much of the Unix overhead is related to network communication. Only a small portion of the overhead is spent in executing code in the TreadMarks library. We conclude therefore that the largest single overhead contribution stems from network communication or related events, validating our focus on reducing the number of messages and the amount of data exchanged. Space overhead consists of memory used for twins, diffs, and other TreadMarks data structures. In the current version of the system, 4 megabytes of memory are statically allocated for diffs and 0.5 megabyte for other data structures. A garbage collection procedure is invoked if these limits are exceeded. Space for twins is dynamically allocated. For a representative example of a large ILINK run, namely the data set with a sequential running time of 4,085 seconds, the maximum memory usage for twins at any point in the execution was approximately 1 megabyte per processor.

9 Related Work

Our goal in this section is not to provide an extensive survey of parallel programming research, but instead to illustrate alternative approaches. We present one example system for each of these approaches. We first discuss alternative programming models, and then turn to different implementations of the shared memory programming models.

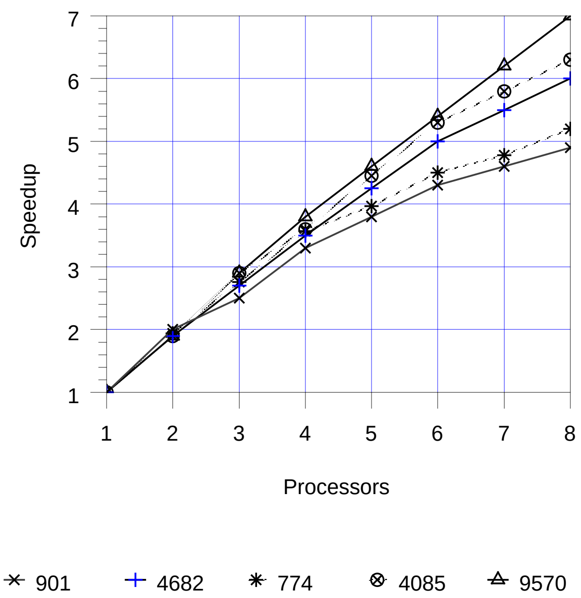

Figure 10 Speedup Results for ILINK: Each line represents a different data set. The numbers at the bottom indicate the sequential execution time in seconds for the corresponding data set.

9.1 Alternative Programming Models

Message Passing (PVM). Currently, message passing is the prevailing programming paradigm for distributed memory systems. Parallel Virtual Machine (PVM) [12] is a popular software message passing package. It allows a heterogeneous network of computers to appear as a single concurrent computational engine. TreadMarks is currently restricted to a homogeneous set of nodes. While programming in PVM is much easier and more portable than programming in the native message passing paradigm of the underlying machine, the application programmer still needs to write code to exchange messages explicitly. The goal in TreadMarks is to remove this burden from the programmer. For programs with complex data structures and sophisticated parallelization strategies, we believe this to be a major advantage. The genetic linkage program provides a compelling example of this argument. We built both a TreadMarks and PVM implementation of ILINK. In each ILINK iteration, each process updates a subset of a large and sparse array of probabilities. To implement this using message passing requires additional code to remember which locations were updated and to marshal and unmarshal the modified values from memory to a message and vice versa. In TreadMarks the shared memory mechanism transparently moves these values between processors as needed. On the downside, message passing implementations can be more efficient than shared memory implementations. Returning to the ILINK example, whereas in PVM all updates are sent in a single message, the TreadMarks ILINK program takes several page faults and an equal number of message exchanges to accomplish the same goal. For a more detailed comparison in programmability and performance between TreadMarks and PVM we refer the reader to Lu's M.S. thesis, which includes a comparison for nine different applications [10].

Implicit Parallelism (HPF). TreadMarks and PVM are both explicitly parallel programming methods: the programmer has to divide the computation among different threads and use either synchronization or message passing to control the interactions among the concurrent threads. With implicit parallelism, as in HPF [6], the user writes a single-threaded program, which is then parallelized by the compiler. In particular, HPF contains data distribution primitives, which may be used by the compiler to drive the parallelization process. This approach is suitable for data-parallel programs, such as Jacobi. The memory accesses in these programs are regular and can be determined completely at compile time. For these applications, the compiler can typically produce code that runs more efficiently than the same application coded for DSM, because the compiler can predict accesses while the DSM system can only react to them. Recent work has explored extensions of this paradigm to irregular computations, often involving sparse arrays, such as in ILINK. Programs exhibiting dynamic parallelism, such as TSP or MIP, are not easily expressed in the HPF framework.

9.2 Alternative Distributed Shared Memory Implementations

Hardware Shared Memory Implementations (DASH). An alternative approach to shared memory is to implement it in hardware, using a snooping bus protocol for a small number of processors or using a directory-based protocol for larger number of processors (e.g., [8]). We share with this approach the programming model, but our implementation avoids expensive cache controller hardware. On the other hand, a hardware implementation can efficiently support applications with finer-grain parallelism. We have some limited experience with comparing the performance of hardware and software shared memory [5]. In particular, we compared the performance of four applications, including a slightly older version of ILINK, on an 8-processor SGI 4D/380 hardware shared memory multiprocessor and on TreadMarks running on our 8-processor ATM network of

DECStation-5000/240s. An interesting aspect of this comparison is that both systems have the same processor, running at the same clock speed and with the same primary cache, and both use the same compiler. Only the provision for shared memory is different: a hardware bus-based snoopy protocol on the SGI, and a software release-consistent protocol in TreadMarks. Identical programs were used on the SGI and on TreadMarks. For the ILINK data sets with long running times, the communication-to-computation ratio is small, and the differences were minimal. For the shorter runs, the differences became more pronounced.

Sequentially-Consistent Software Distributed Shared Memory (IVY). In sequential consistency, messages are sent, roughly speaking, for every write to a shared memory page for which there are other valid copies outstanding. In contrast, in release consistency, messages are sent for every synchronization operation. Although the net effect is somewhat application dependent, release consistent DSMs in general send fewer messages than sequentially consistent DSMs and therefore perform better. A recent paper by Carter et al. [3] contains a comparison of seven application programs run either with eager release consistency (Munin) or with sequential consistency. Compared to a sequentially consistent DSM, Munin achieves performance improvements ranging from a few to several hundred percent, depending on the application.

Lazy vs. Eager Release Consistency (Munin). Lazy release consistency causes fewer messages to be sent than eager release consistency, as implemented in Munin [3]. At the time of a lock release, Munin sends messages to all processors who cache data modified by the releasing processor. In contrast, in lazy release consistency, consistency messages only travel between the last releaser and the new acquirer. Lazy release consistency is somewhat more complicated than eager release consistency. After a release, Munin can forget about all modifications the releasing processor made prior to the release. This is not the case for lazy release consistency, since a third processor may later acquire the lock and need to see the modifications. In practice, our experience indicates that for networks of workstations, in which the cost of sending messages is high, the gains achieved by reducing the number of messages outweighs the cost of a more complex implementation. In particular, Keleher has compared the performance of ten applications under lazy and eager release consistency, and found that for all but one (3-D Fast Fourier Transform) the lazy implementation performed better [5]. It was also shown that an invalidate protocol works better than an update protocol, because of the large amount of data resulting from the update protocol.

Entry Consistency (Midway). Entry consistency is another relaxed memory model [2]. As in release consistency, consistency actions are taken in conjunction with synchronization operations. Unlike release consistency, however, entry consistency requires that each shared data object be associated with a synchronization object. When a synchronization object is acquired, only the modified data associated with that synchronization object is made consistent. Since there is no such association in release consistency, it has to make all shared data consistent. As a result, entry consistency generally requires less data traffic than lazy release consistency. The entry consistency implementation in Midway also uses an update protocol, unlike TreadMarks which uses an invalidate protocol. The programmability and performance differences between these two approaches are not yet well understood.

Structured DSM Systems (Linda). Rather than providing the programmer with a shared memory space organized as a linear array of bytes, structured DSM systems offer a shared space of objects or tuples [1], which are accessed by properly synchronized methods. Besides the advantages from a programming perspective, this approach allows the compiler to infer certain optimizations that can be used to reduce the amount of communication. For instance, in the TSP example, an object-oriented system can treat the queue of partial tours as a queue object with enqueue and dequeue operations. Similarly, in Linda, these operations would be implemented by means of the `in` and `out` primitives on the tuple space. These operations can typically be implemented more efficiently than paging in the queue data structure as happens in DSM. On the downside, the objects that are "natural" in the sequential program are often not the right grain of parallelization, requiring more changes to arrive at an efficient parallel program. With sparse updates of larger arrays, as in ILINK for instance, there is little connection between the objects in the program and the updates.

10 Conclusions and Further Work

Our experience demonstrates that with suitable implementation techniques, distributed shared memory can provide an efficient platform for parallel computing on networks of workstations. Large applications were ported to the TreadMarks distributed shared memory system with little difficulty and good performance. In our further work we intend to experiment with additional real applications, including a seismic modeling code. We are also developing various tools to further ease the programming burden and improve performance. In particular, we are investigating the use of compiler support for prefetching and the use of performance monitoring tools to eliminate unnecessary synchronization.

References

[1] S. Ahuja, N. Carreiro, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[2] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the '93 CompCon Conference*, pages 528–537, February 1993.

[3] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related information in distributed shared memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[4] S.K. Gupta, A.A. Schäffer, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. Integrating parallelization strategies for linkage analysis. *Computers and Biomedical Research*, 28:116–139, June 1995.

[5] P. Keleher. *Distributed Shared Memory Using Lazy Release Consistency.* PhD thesis, Rice University, December 1994. Appeared as Rice Technical Report RICE COMP-TR-240 and available by anonymous ftp from cs.rice.edu under public/TreadMarks/papers.

[6] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, 1994.

[7] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[8] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[9] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[10] H. Lu. Message passing versus distributed shared memory on networks of workstations. Master's thesis, Rice University, April 1995. Appeared as Rice Technical Report RICE COMP-TR-250 and available by anonymous ftp from cs.rice.edu under public/TreadMarks/papers.

[11] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, 24(5):54–64, May 1990.

[12] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency:Practice and Experience*, 2(4):315–339, December 1990.