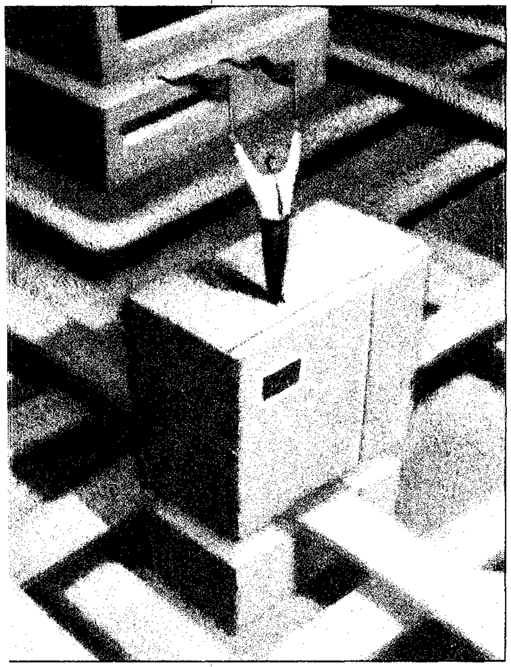


TreadMarks: Shared Memory Computing on Networks of Workstations

Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel
Rice University

Shared memory facilitates the transition from sequential to parallel processing. Since most data structures can be retained, simply adding synchronization achieves correct, efficient programs for many applications.

High-speed networks and improved microprocessor performance are making networks of workstations an appealing vehicle for parallel computing. By relying solely on commodity hardware and software, networked workstations can offer parallel processing at a relatively low cost.

A network-of-workstations multiprocessor can be realized as a processor bank in which dedicated processors provide computing cycles, or it can consist of a dynamically varying set of machines that perform long-running computations during idle periods. In the latter case, the hardware cost is essentially zero, since many organizations already have extensive workstation networks.

In terms of performance, networked workstations approach or exceed supercomputer performance for some applications. These loosely coupled multiprocessors will by no means replace the more tightly coupled designs, which, because of lower latencies and higher bandwidths, are more efficient for applications with stringent synchronization and communication requirements. However, advances in networking technology and processor performance are expanding the class of applications that can be executed efficiently on networked workstations.

DSM OVERVIEW

In this article, 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.[1]

Figure 1 illustrates a DSM system consisting of *N* networked workstations, each with its own memory. 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, the "native," or message passing, programming model on workstation networks requires the programmer to specify interprocessor communication—a daunting task for programs with complex data structures and sophisticated parallelization strategies.

In a DSM system, the programmer can focus on developing algorithms instead of managing partitioned data sets and communicating values. Also, since DSM provides the same programming environment as hardware shared-memory multiprocessors, programs written for a DSM system are easily ported to a shared memory multiprocessor. However, porting from a hardware shared-memory multiprocessor to a DSM system may require more modifications because the DSM system's higher latencies put a greater value on locality of memory access.

The programming interfaces to DSM systems may differ in various respects. Here, we focus on the memory *structure* and *consistency model*. An unstructured memory appears as a linear array of bytes, but in a structured memory, processes access memory in terms of objects or tuples. The consistancy model refers to how shared memory updates become visible to the system's processes. The intuitive model 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*, in which all processes see memory as if they were executing on a single multiprogrammed processor.[2] 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; therefore, much research has been done on *relaxed memory models*.

One distinguishing feature of a DSM implementation is whether it uses the virtual-memory page-protection hardware to detect shared memory accesses. The naive use of virtual-memory protection hardware may lead to poor performance because of discrepancies between the machine's page size and the application's granularity of sharing.

Our system, TreadMarks,[3] provides shared memory as a linear array of bytes via a relaxed memory model called release consistency. The implementation uses the virtual memory hardware to detect accesses, but it uses a multiple-writer protocol to alleviate problems caused by mismatches between page size and application granularity.

TreadMarks runs at the user level on Unix workstations, without kernel modifications or special privileges and with standard Unix interfaces, compilers, and linkers. As a result, the system is fairly portable and has been ported to a number of platforms. These include IBM RS-6000, SP-1, and SP-2; DEC Alpha and DECStation, as well as Hewlett-Packard, Silicon Graphics, and Sun systems.

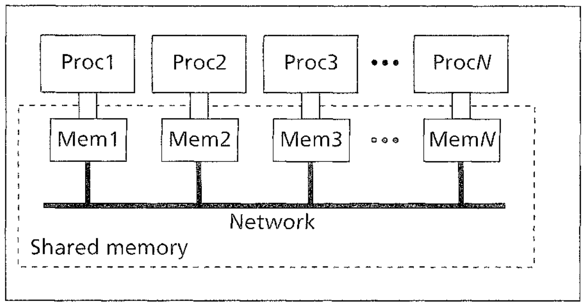


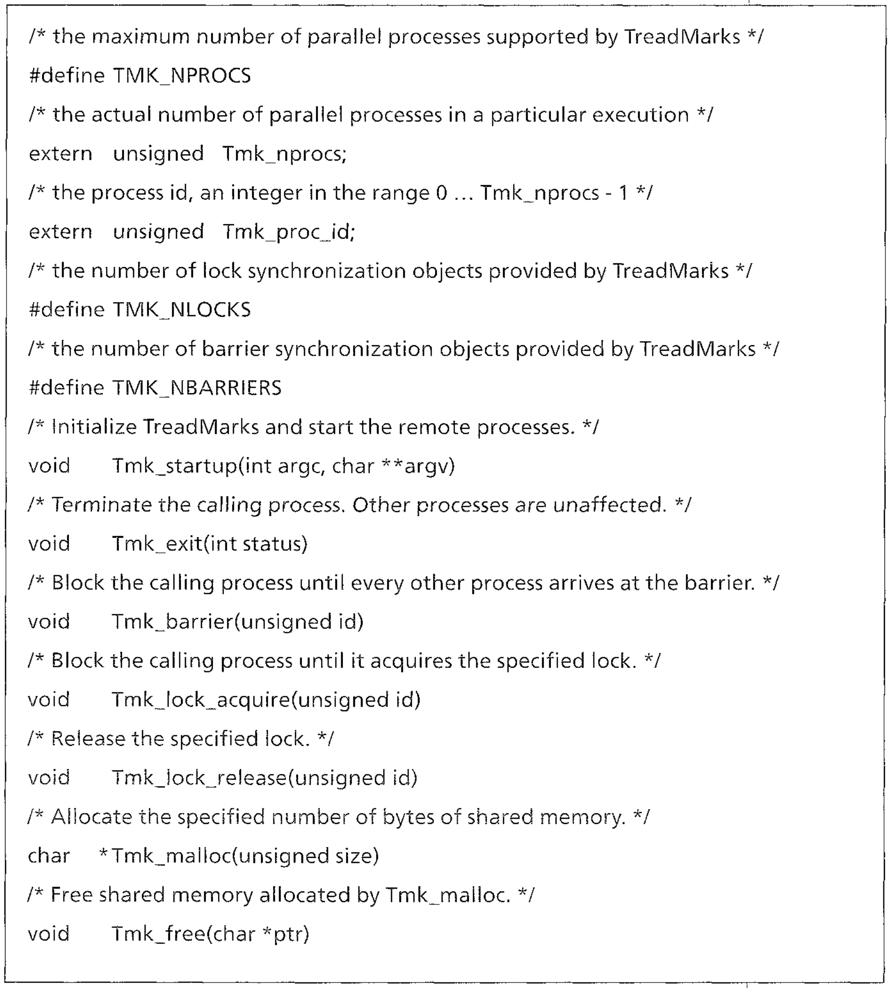
Figure 1. Distributed shared memory: Each processor sees a shared address space, denoted by the dashed outline, rather than a collection of distributed address spaces.

SHARED MEMORY PROGRAMMING

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. Memory allocated by Tmk_malloc() is shared. Memory allocated statically or by a call to malloc() is private to each process.

Understanding the purpose of the synchronization



```
/* 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.

primitives is essential to programming with TreadMarks. Synchronization primitives let the programmer specify ordering constraints between different processes' *conflicting* shared-memory accesses. Two shared-memory accesses are said to conflict if they are issued by different processors to the same memory location and at least one access is a write. Parallel programs are subject to data races, a bug that makes execution timing-dependent, if there is no synchronization between two conflicting accesses. For example, if one access is a read and the other is a write, other execution orders may cause different outcomes for each execution. Data races can be avoided by introducing synchronization.

TreadMarks provides two synchronization primitives: Barriers and locks. Barriers are global. Locks can be used to implement critical sections.

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 arrive at the same barrier. A Tmk_lock_acquire() call acquires a lock for a calling process, and Tmk_lock_release() releases it. No process can acquire a lock while another is holding it. A lock can be used to implement critical sections. However, these particular synchronization primitives are not fundamental to TreadMarks' design, and we may add other primitives later.

Two simple illustrations

Two simple problems (larger applications are discussed later) illustrate the TreadMarks API. Jacobi iteration (Figure 3) shows the use of barriers, and the traveling salesman problem (Figure 4) shows the use of locks.

JACOBI ITERATION. 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 new values to avoid overwriting an element's old value before it is used by its neighbor. In the parallel version, all processors are assigned roughly equal-size bands of rows. Neighboring processes share the rows on a band's boundary.

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

TRAVELING SALESMAN PROBLEM. TSP uses a simple branch-and-bound algorithm to find the shortest route that starts at a designated city, passes through every other city on the map once, and returns to the original city. The program maintains the length of the shortest route found so far in Shortest_length. Partial routes are expanded one city at a time. If the current length of a partial route plus a lower bound on the remaining portion is longer than the current shortest tour, that route is not explored further, because it cannot lead to a shorter total. The lower bound is computed by a fast, conservative approximation of the length of the minimum spanning tree connecting all nodes not yet in the trip with the current route's first and last nodes.

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 adds partial tours until a partial tour longer than a threshold number of cities reaches the top of the queue. It removes this partial tour and tries all permutations of the remaining cities. Next, the program compares the shortest tour including this partial tour with the current shortest tour and, if necessary, updates it. Finally, the program returns to the tour queue and tries again to remove a promising partial tour of sufficient length.

Figure 4 shows pseudocode for the parallel TreadMarks TSP program. Process 0 allocates the shared data structures (the queue and the minimum length). Exclusive access is achieved by surrounding all accesses to these shared data structures by a lock acquire and a lock release. All processes wait at Tmk_barrier(0) to ensure proper initialization before computation. Each process then acquires the queue lock to find a promising partial tour that is long enough to expand 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. This process continues until the queue is empty.

IMPLEMENTATION CHALLENGES

DSM systems can either migrate or replicate data to implement the shared memory abstraction. Most DSM systems replicate data because that approach gives the best performance for a wide range of application parameters.[4] With replicated data, memory consistency is central to the system, since the DSM software must control replication in a manner that provides a single shared-memory abstraction.

The *consistency model* defines the expected memory behavior for the programmer. The first DSM system, Ivy,[1] implemented *sequential consistency*.[2] In this memory model, processes observe shared memory as if they were executing on a multiprogrammed uniprocessor with a single memory. In other words, all memory accesses are totally ordered, and the order is compatible with the program's memory access order in each process.

Ivy's implementation of sequential consistency uses the virtual memory hardware to maintain memory consistency.

```
#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. Pseudocode for the TreadMarks Jacobi program.

```
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 Queue 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. Pseudocode for the TreadMarks traveling salesman program.

The local (physical) memories of each processor form a cache on the global virtual address space (see Figure 5). When a page is not in a processor's local memory, 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. Figure 5 shows the activity for a page fault at processor 1, which results in retrieval of a copy from processor 3's local memory. For a read fault, the page is replicated with read-only access for all replicas. For a write fault, an invalidate message is sent to all processors with copies of the page. Each processor receiving this message invalidates its copy and sends an acknowledgment 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 extensive communication, which is expensive on a workstation network. Sending a message can involve traps into the operating system kernel, interrupts, context switches, and execution of several networking software layers. Therefore, the number of messages and the amount of data exchanged must be kept low.

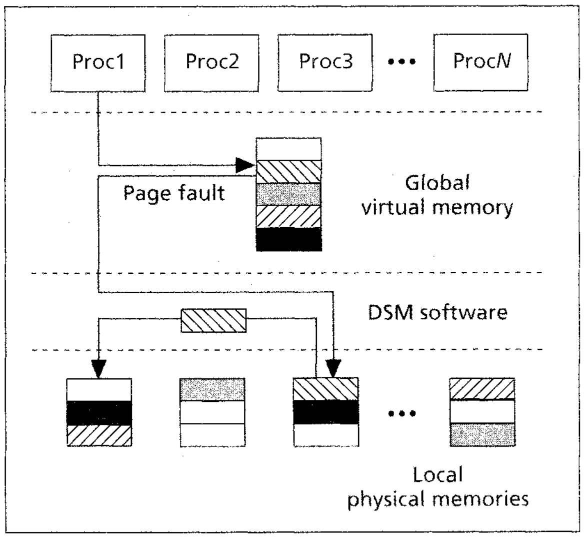


Figure 5. Operation of the Ivy DSM system.

Ivy encountered several communication problems. For example, updating the current shortest tour in the traveling salesman problem sent invalidations to all other processors that cache the page containing the current shortest tour. However, since this variable is accessed only within the critical section protected by the corresponding lock, it suffices to send an invalidation only to the next processor acquiring the lock and only at the time of lock acquisition.

Another problem involved false sharing, which 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 can be common. The Jacobi program in Figure 3 suffers from false sharing if the grid array is laid out so that portions allocated to different processors lie within the same page. As both processors update their portion of the grid array, they write concurrently to the same page. Assume that initially processor P_1 holds the sole writable copy. 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 next writes to the page, the same sequence occurs with P_1 and P_2 interchanged. As each process writes to the page while it is held by the other process, the page travels across the network. This repeated back-and-forth transmission is often called the "ping-pong effect."

To address these problems, we experimented with the following relaxed consistency models and protocols.

Synchronization eliminates the need to make any shared memory update from one process visible to another process before they synchronize with each other.

LAZY RELEASE CONSISTENCY

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. This synchronization eliminates the need to make any shared memory update 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.

To illustrate this principle, we will use the Jacobi and TSP examples (Figures 3 and 4). In Jacobi, writes to shared memory occur after barrier 1 is passed, when the newly computed values are copied from the scratch array to the grid array. This phase of the computation terminates when barrier 2 is passed. Barrier 2 prevents processes from starting the next iteration before all new values are written to the grid array, and it is essential for correctness (to avoid data races) regardless of the memory model. However, its presence lets us delay notifying a process about another process's updates 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 and inserting them in the queue. Updates to the task queue structure require a series of shared memory writes regarding task size and so on. Atomic access to the task queue data structure is necessary for correct program execution. Only one processor is permitted access to the task queue at a time. This is guaranteed by a lock acquire and release around these operations. To access the tour queue, a process must acquire the lock. It therefore suffices to inform the next process acquiring the lock of the changes to the tour queue, which can be done when the lock is acquired.

These two examples illustrate release consistency's underlying principle. Synchronization is introduced in a shared memory parallel program to prevent processes from accessing certain memory locations before the synchronization operation completes. From that it follows that a process does not need to be informed of modifications until the synchronization operation completes. If the program is free of data races, it will appear as if it is executing on a sequentially consistent memory, the intuitive memory model that programmers expect. The above is true on one condition: All synchronization must use TreadMarks-supplied primitives. Otherwise, TreadMarks cannot tell when to make shared memory consistent.

Release consistency implementations

The release consistency definition specifies the latest time a shared memory update can become visible to a particular processor. This allows considerable latitude in when and how shared memory updates are propagated. TreadMarks uses the lazy release consistency algorithm.[3] Roughly speaking, this enforces consistency at acquire time, in contrast to Munin's[5] earlier version of release consistency, sometimes called "eager release consistency," which enforces consistency at release time.

Figure 6 illustrates the principal advantage of lazy release consistency. Assume x is replicated at all processors. With eager release consistency, a message must be sent to all processors 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, thus reducing message traffic. Lazy release consistency also allows piggybacking the notification onto the lock grant message from the releasing to the acquiring process.

Communicating that a page has changed and communicating the changed values within the page are distinct operations in TreadMarks. To perform the latter, TreadMarks uses an *invalidate* protocol. A modified page is invalidated after an acquire. A later access causes an access miss, which in turn causes installation of an up-to-date copy of the page. An alternative method would be an update protocol in which the acquire message contains the new values. (See Keleher[3] for details on TreadMarks protocols.)

MULTIPLE-WRITER PROTOCOLS

Most hardware cache and DSM systems use single-writer protocols. These protocols let multiple readers access a page simultaneously, but a writer must have sole access before performing modifications. Single-writer protocols are easy to implement because all copies of a page are always identical, and page faults can be satisfied by retrieving a page from any processor that 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 cause access misses if processors are still accessing the page. Also, false sharing can degrade performance even more because of interference between unrelated accesses. DSM systems typically suffer much more from false sharing than 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 processes P_1 and P_2 concurrently write to different locations within the same page and that both initially have an identical valid copy. TreadMarks uses the virtual memory hardware to detect modifications (see Figure 7). The shared page is initially write-protected. When P_1 writes to the page, TreadMarks creates a copy, 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 can occur without software intervention. When P_1 arrives at the barrier, we now have the modified copy and the unmodified twin. A word-by-word comparison creates a diff, a run-length encoding of the page modifications. Once the diff has been created, the twin is discarded. The same sequence of events occurs on P_2.

Since these events are local to each processor, they do not require the message exchanges of a single-writer protocol. When P_1 and P_2 synchronize (through a barrier, for instance), P_1 is informed that P_2 has modified the page, and vice versa, and both invalidate their copies. When they later access the page, 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. The same sequence of events happens on P_2. Thus, except for initial accesses, pages are updated exclusively by applying diffs, and complete new copies are never needed.

The primary benefit of diffs is that they can be used to implement multiple-writer protocols, thereby reducing the effects of false sharing. In addition, diffs significantly reduce overall bandwidth requirements because they are typically much smaller than a page.

One might wonder what happens when two processes modify overlapping portions of a page. This corresponds to a data race; two processes are writing to the same location without intervening synchronization. Therefore, it is almost certainly a program error. Even on a sequentially consistent memory, the outcome would be timing-dependent. The same is true in TreadMarks. We could modify TreadMarks to check for such occurrences but have not yet done so.

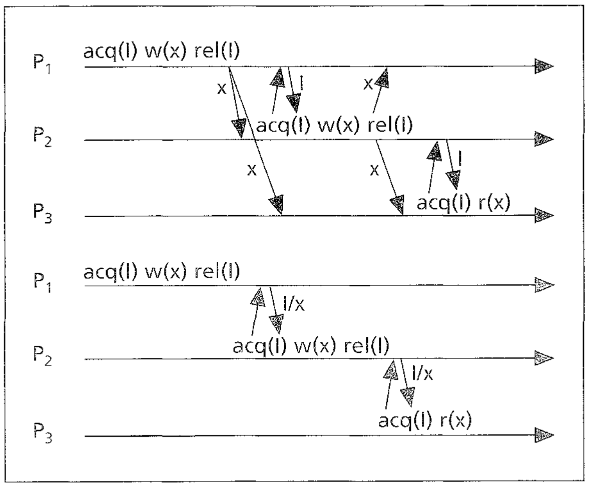


Figure 6. Eager release consistency (top) versus lazy release consistency (bottom). The figure depicts the execution of three processes, P_1, P_2, and P_3, with the time axis going from left to right. The processes acquire and release the lock l and read and write the variable x.

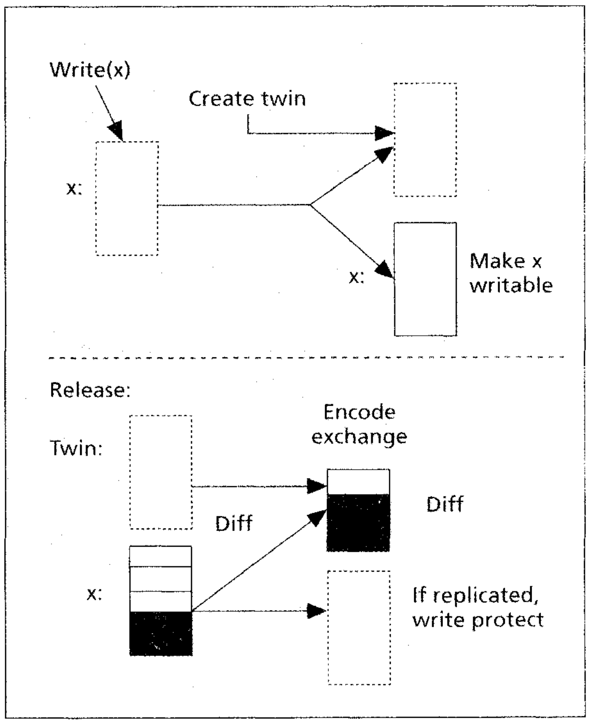


Figure 7. Diff creation.

TREADMARKS SYSTEM

TreadMarks is implemented entirely as a user-level library on top of Unix. Kernel modifications are unnecessary because modern Unix implementations provide all required communication and memory management functions. 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.

Operation

TreadMarks implements intermachine communication using UDP/IP through the Berkeley sockets interface. Since UDP/IP does not guarantee reliable delivery, TreadMarks uses lightweight, operation-specific, user-level protocols to ensure message arrival. Every message sent by TreadMarks is a request or a response. Request messages are sent 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 response message arrives. If no response arrives within a certain time, the original request is retransmitted. To minimize delay 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 receiving a request message, the handler 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 gain restricted access to a shared page generates a SIGSEGV signal. The SIGSEGV signal handler examines local data structures to determine page 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 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 creates a twin from the pool of free pages. It takes the same action in response to a fault

Related work and comparisons

Alternative programming models

Message passing (PVM)

Message passing is the prevailing programming paradigm for distributed memory systems. In Parallel Virtual Machine (PVM),[1] a popular message-passing software package, a heterogeneous network of computers appears 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 underlying machine's native message-passing paradigm, the application programmer still needs to write code to exchange messages explicitly. Since TreadMarks removes that burden, it is especially advantageous for programs with complex data structures and sophisticated parallelization strategies, such as the genetic linkage program described in the main text.

For the genetic linkage program, we built both a TreadMarks and PVM implementation of ILink. In each ILink iteration, each process updates a subset of a large, sparse array of probabilities. A message-passing implementation 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. TreadMarks' shared memory mechanism transparently moves these values between processors as needed.

However, message-passing implementations can be more efficient. In the ILink example, PVM sends all updates in a single message, while TreadMarks takes several page faults and an equal number of message exchanges to accomplish the same goal. (See Lu[2] for a detailed comparison of TreadMarks and PVM programmability and performance for nine applications.)

Implicit parallelism (HPF)

TreadMarks and PVM are both explicitly parallel programming methods: The programmer must divide the computation among different threads and use either synchronization or message passing to control interactions among concurrent threads. With implicit parallelism, as in HPF,[3] the user writes a single-threaded program, which is then parallelized by the compiler. In particular, HPF contains data distribution primitives, which the compiler can use to drive the parallelization process. This approach is suitable for data-parallel programs, such as Jacobi, with regular memory accesses that can be determined at compile time. For these applications, the compiler typically produces more efficient code. It can predict accesses, while a DSM system can only react to them.

Programs exhibiting dynamic parallelism are not easily expressed in the HPF framework, but extensions, often involving sparse arrays, have been explored.

Alternative DSM implementations

Hardware shared-memory implementations (DASH)

An alternative approach is to implement shared memory in hardware, using a snooping bus protocol for a small number of processors or a directory-based protocol for more processors.[4] This approach requires cache controller hardware, but it also efficiently supports applications with finer grain parallelism.

We have compared the hardware and software shared-memory performance of four applications,[5] including a slightly older version of ILink on an eight-processor SGI 4D/480 hardware shared-memory multiprocessor and on TreadMarks running on our eight-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. The programs were identical; only the shared memory—a hardware bus-based snoopy protocol on the SGI and a software release-consistent protocol in TreadMarks—was different. For ILink data sets with long runtimes, the communication-to-computation ratio is small, and the differences were minimal. For shorter runs, the differences are 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 that has other valid copies outstanding. In release consistency, messages are sent for every synchronization operation. Although somewhat application-dependent, release-consistent DSMs generally send fewer messages than sequentially consistent DSMs and, therefore, perform better. In Carter et al.'s[6] comparison of seven programs run with caused by a write to a page in read-only mode. Finally, the handler upgrades access rights to the original page and returns.

Costs

Our experimental environment consists of eight DECstation-5000/240s, with a 4-kilobyte page size, running Ultrix V4.3. Each machine has a Fore ATM TCA-100 interface connected to a Fore ATM ASX-100 switch. The connection between the interface boards and the switch operates at 100 Mbps; the switch has an aggregate throughput of 1.2 Gbps. Unless otherwise noted, the performance numbers describe eight-processor executions on the ATM LAN using the low-level adaptation layer protocol AAL3/4.

The minimum roundtrip time using send and receive for the smallest possible message is 500 microseconds. Using a signal handler to receive the message at one processor increases the roundtrip time to 670 microseconds.

The minimum time to remotely acquire a free lock is 827 microseconds. The minimum time to perform an eight-processor barrier is 2,186 microseconds. A remote access miss, to obtain a full page from another processor, takes 2,792 microseconds.

Munin's eager release consistency or with sequential consistency, using Munin improved performance from a few to several hundred percent depending on the application.

Lazy versus eager release consistency (Munin)

Lazy release consistency results in fewer messages than eager release consistency as implemented in Munin.[5] Upon a lock release, Munin sends messages to all processors that cache data modified by the releasing processor. With lazy release consistency, messages travel only between the last releaser and the new acquirer. However, lazy release consistency is more complicated than eager release consistency. After a release, Munin can forget all prior modifications made by the releasing processor. This is not the case for lazy release consistency, since a third processor may later acquire the lock and need to see the modifications.

Our experience indicates that for workstation networks, where message costs are high, the gains achieved by message reduction outweigh the cost of more complex implementation. In particular, Keleher's[5] comparison of 10 applications under lazy and eager release consistency showed better performance for the lazy implementation on all but one, a 3D fast Fourier transform. It also showed that an invalidate protocol works better than an update protocol, which results in a large amount of data movement.

Entry consistency (Midway)

Entry consistency is another relaxed memory model.[7] Like release consistency, it combines consistency actions with synchronization operations, but unlike release consistency, it requires associating each shared data object with a synchronization object. When a synchronization object is acquired, only the modified data associated with it is made consistent. Since there is no such association in release consistency, it has to make all shared data consistent. Thus, entry consistency generally requires less data traffic than lazy release consistency. The Midway entry consistency implementation also uses an update protocol, while TreadMarks 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[8] accessed by properly synchronized methods. Besides the advantages from a programming perspective, this approach allows the compiler to infer certain optimizations that can reduce communication. For instance, in the traveling salesman problem, 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 in DSM systems. On the downside, natural objects in the sequential program are seldom the right grain of parallelism, so efficient parallelization requires more changes. With sparse updates of larger arrays, as in ILink, there is little connection between program objects and updates.

References

1. V. Sunderam, "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience*, Vol. 2, No. 4, Dec. 1990, pp. 315-339.
2. H. Lu, "Message Passing versus Distributed Shared Memory on Networks of Workstations," master's thesis, Rice University, Tech. Report Rice Comp-TR-250, ftp cs.rice.edu under public/TreadMarks/papers.
3. C. Koelbel et al., *High Performance Handbook*, MIT Press, Cambridge, Mass., 1994.
4. D. Lenoski et al., "Directory-Based Cache Coherence Protocol for the Dash Multiprocessor," *Proc. 17th Annual Int'l Symp. Computer Archictecture*, IEEE Computer Society Press, Los Alamitos, Calif., 1990, pp. 148-159.
5. P. Keleher, "Distributed Shared Memory Using Lazy Release Consistency," PhD dissertation, Rice University, Tech. Report Rice Comp-TR-240, ftp cs.rice.edu under public/TreadMarks/papers, 1994.
6. J.B. Carter, J.K. Bennett, and W. Zwaenepoel, "Techniques for Reducing Consistency-Related Information in Distributed Shared Memory Systems," *ACM Trans. Computer Systems*, Vol. 13, No. 3, Aug. 1995, pp. 205-243.
7. B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon, "Midway Distributed Shared Memory System," *Proc. Compcon 93*, IEEE Computer Society Press, Los Alamitos, Calif., 1993, pp. 528-537.
8. S. Ahuja, N. Carreiro, and D. Gelernter, "Linda and Friends," *Computer*, Vol. 19, No. 8, Aug. 1986, pp. 26-34.

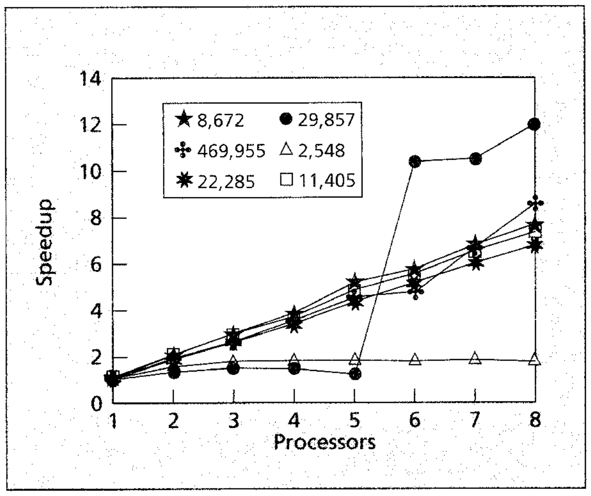


Figure 8. Speedup results for MIPLIB problems. Each line represents a different data set. The numbers in the legend indicate the sequential execution in seconds for the corresponding data set. Only data sets with sequential running times over 2,000 seconds are presented.

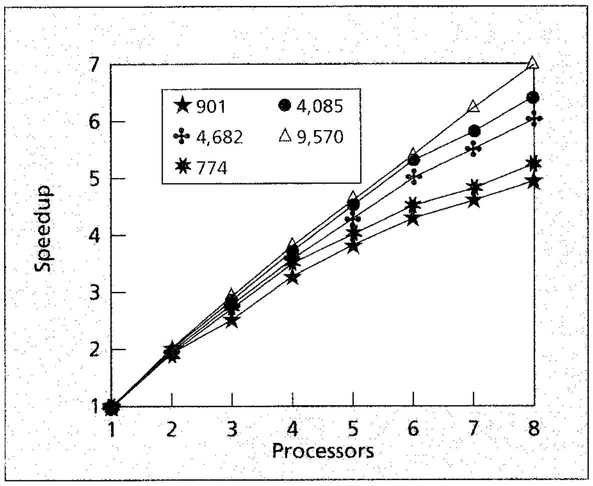


Figure 9. Speedup results for ILink. Each line represents a different data set. The numbers in the legend indicate the sequential execution time in seconds for the corresponding data set.

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. For the worst case, when every other word in the page is changed, making a diff takes 686 microseconds.

APPLICATIONS

We have implemented a number of applications in TreadMarks and reported some benchmark results earlier.[3] Here, we describe our experience with two large, recently implemented applications, mixed integer programming and genetic linkage analysis. These applications were parallelized, starting from an existing, efficient sequential code. Modification to arrive at an efficient parallel code proved to be relatively minor.

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). The precise mathematical formulation is

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

The TreadMarks MIP code takes a branch-and-cut approach. The MIP problem is first relaxed to the corresponding LP problem. The LP solution generally produces noninteger values for some 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 that $x_i \geq \lceil x_i \rceil$. Over time, the algorithm generates a tree of such branches. As soon as the algorithm finds an LP solution that satisfies the integer constraints, this solution establishes a bound on the objective function's final value, and nodes for which the LP result is inferior are not explored further. 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 uses cutting planes as additional constraints to tighten the problem description. Separate locks protect the two shared data structures: the current best solution and the MIP problem queue.

We used the code to solve all 51 of the MIPLIB library's problems, which include representative examples from airline crew scheduling, network flow, plant location, and fleet scheduling. Figure 8 shows the speedups obtained for problems with sequential running times 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 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 preprocessing step, which is not yet parallelized. The code was also used to solve a previously unsolved multicommodity flow problem. The problem took roughly 28 CPU days on an eight-processor IBM SP-1 and also exhibited near-linear speedup.

Genetic linkage analysis

Genetic linkage analysis is a statistical technique that uses family pedigree information to map 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 parent's chromosomes. In reality, inheritance is more complicated due to recombination, in which the child's chromosome receives a piece of both strands. 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, we can compute the gene's approximate location on the chromosome.

We parallelized ILink, a widely used genetic linkage analysis program that is part of the Fastlink package.[6] ILink takes as input a family tree, called a pedigree, augmented with some genetic information about family members. 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 known genetic information of family members. For each nuclear family member, the algorithm updates a large array of conditional probabilities. Each represents the probability that the individual has certain genetic characteristics, conditioned on θ and on the part of the family tree already traversed.

The algorithm is parallelized by splitting the iteration space per nuclear family to balance the load among the available processors. Load balancing is essential and relies on knowledge of the genetic information represented in the array elements. An alternative approach, splitting 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 about whom little genetic information is known).

Figure 9 presents speedups for various data sets 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 of the greater communication-to-computation ratio. 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 overhead—that is, 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 executing code in the TreadMarks library. The largest single overhead stems from network communication or related events, which validates our focus on reducing messages and data exchange. Space overhead consists of memory used for twins, diffs, and other TreadMarks data structures. The current system statically allocates 4 megabytes of memory 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.

OUR EXPERIENCE DEMONSTRATES THAT with suitable implementation techniques, distributed shared memory can provide an efficient platform for parallel computing on networked workstations. We ported large applications to the TreadMarks DSM system with little difficulty and good performance. We intend to experiment with additional 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 compiler support for prefetching and performance monitoring tools to eliminate unnecessary synchronization. ▮

Acknowledgments

This research was supported in part by the National Science Foundation under Grants CCR-9116343, CCR-9211004, CDA-9222911, BIR-9408503, and CDA-9502791, and by the Texas Advanced Technology Program and Tech-Sym Inc. under Grant 003604012.

References

1. K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. Computer Systems*, Vol. 7, No. 4, Nov. 1989, pp. 321-359.
2. L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Trans. Computers*, Vol. C-28, No. 9, Sept. 1979, pp. 690-691.
3. P. Keleher, "Distributed Shared Memory Using Lazy Release Consistency," PhD dissertation, Rice University, Tech. Report Rice Comp-TR-240, ftp cs.rice.edu under public/TreadMarks/papers, 1994.
4. M. Stumm and S. Zhou, "Algorithms Implementing Distributed Shared Memory," *Computer*, Vol. 24, No. 5, May 1990, pp. 54-64.
5. J.B. Carter, J.K. Bennett, and W. Zwaenepoel, "Techniques for Reducing Consistency-Related Information in Distributed Shared Memory Systems," *ACM Trans. Computer Systems*, Vol. 13, No. 3, Aug. 1995, pp. 205-243.
6. S.K. Gupta et al., "Integrating Parallelization Strategies for Linkage Analysis," *Computers and Biomedical Research*, Vol. 28, June 1995, pp. 116-139.

***Cristiana Amza** is a PhD student in the Department of Computer Science at Rice University. Her interests are in all aspects of distributed systems and parallel computing, in particular, distributed shared memory and the use of new advances in network technology to improve distributed shared memory performance. She received the BS degree in computer science from Bucharest Polytechnic Institute, Bucharest, Romania, in 1991.*

***Alan L. Cox** is an assistant professor in the Department of Computer Science at Rice University. His research interests include cache coherence protocols and data placement for shared memory multiprocessors and distributed shared memory for workstation networks. He was named an NSF Young Investigator in 1994. He received the BS degree in applied mathematics from Carnegie Mellon University in 1986 and MS and PhD degrees in computer science from the University of Rochester in 1988 and 1992.*

***Sandhya Dwarkadas** is a research scientist in the Department of Computer Science at Rice University. Her research interests include parallel and distributed systems, parallel computer architecture, parallel computation, simulation methodology, and performance evaluation. She received the BTech. degree in electrical and electronics engineering from the Indian Institute of Technology, Madras, India, in 1986, and the MS and PhD degrees in electrical and computer engineering from Rice University in 1989 and 1993.*

***Pete Keleher** is an assistant professor in the Department of Computer Science at the University of Maryland, College Park. His research interests include abstractions of coherence, distributed shared memory, operating systems, and parallel computer architecture. He received the BS degree in electrical engineering, and the MS and PhD degrees in computer science from Rice University in 1986, 1993, and 1995.*

***Honghui Lu** is a PhD student in the Electrical and Computer Engineering Department at Rice University. Her research interests include parallel and distributed systems, parallel computation, and performance evaluation. She received the BS degree in computer science and engineering from Tsinghua University, China, in 1992, and the MS degree in electrical and computer engineering from Rice University in 1995.*

***Ramakrishnan Rajamony**, a PhD student in the Department of Electrical and Computer Engineering at Rice University, is currently working on performance debugging of shared memory parallel programs, providing prescriptive feedback to the user. He has also worked on software distributed shared memory systems and compiler schemes for software cache coherence. He received the BTech degree in electrical engineering from the Indian Institute of Technology, Madras, in 1989, and the MS degree in computer engineering from North Carolina State University in 1991.*

***Weimin Yu** is a PhD student in the Department of Computer Science at Rice University, working under the direction of Professor Alan Cox. His research interests include distributed operating systems and distributed programming environments. He received the BE degree in computer science and engineering from Tsinghua University, China, in 1991, and the MS degree in computer science from Rice University in 1994.*

***Willy Zwaenepoel** is a professor in the Department of Computer Science at Rice University. His research interests are in distributed operating systems, fault tolerance, parallel computation, and nonvolatile memory. He received the BS in electrical engineering from the University of Gent, Belgium, in 1979, the MS degree in computer science from Stanford University in 1980, and the PhD degree in electrical engineering from Stanford University in 1984.*

Readers can contact the authors at the Department of Computer Science, Rice University, Houston, TX 77005-1892; e-mail {amza, alc, sandhya}@cs.rice.edu, keleher@cs.umd.edu, {hhl,rrk,weimin, willy}@cs.rice.edu; http://www.cs.rice.edu/~willy/TreadMarks/overview.html. For information on obtaining the TreadMarks system, please send e-mail to treadmarks@ece.rice.edu.

Louise Moser, computer networks area manager for Computer, *coordinated the review of this article and recommended it for publication. Her e-mail address is moser@ece.ucsb.edu.*