TreadMarks: Shared Memory Computing on Networks of Workstations

Christiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher,
Honghui Lu, Ramakrishnan Rajamony, Weimin Yu and Willy Zwaenepoel
Department of Computer Science
Rice University *

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 because most of the data structures can be retained without change. Only synchronization needs to be added. We discuss the techniques used in TreadMarks to provide efficient shared memory, and we discuss the experience with two major 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* [17], 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 [14]. 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 speed and network bandwidth and latency 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 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.

*This research was supported in part by the National Science Foundation under Grants CCR-9116343, CCR-9211004, CDA-9222911, and CDA-9310073, and by the Texas Advanced Technology Program and Tech-Sym Inc. under Grant 003604012.

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 [13]. 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. On 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 programming, DSM provides the same programming environment as that on (hardware) shared-memory multiprocessors, allowing for portability between the two environments. Finally, DSM allows for seamless integration of shared-memory multiprocessor workstations in a network environment.

The actual system described in this paper, TreadMarks [7], runs at user-level on Unix workstations. No kernel modifications or special privileges are required, and standard Unix compilers and linkers are used. The challenge in implementing a DSM system is to make sure that the shared memory abstraction does not result in a large amount of communication. Various techniques have been used in TreadMarks to meet this challenge, including lazy release consistency [6] and multiple writer protocols [3]. This paper first describes the application programming interface provided by TreadMarks (Section 2), Next, we discuss the implementation challenges. (Section 3) and the techniques used to meet these challenges (Sections 4 and 5). We briefly describe the implementation of TreadMarks in Section 6, and we demonstrate its efficiency by discussing our experience with two large applications, mixed integer programming and genetic linkage analysis (Section 7). Finally, we discuss related work in Section 8 and we offer some conclusions and directions for further work in Section 9.

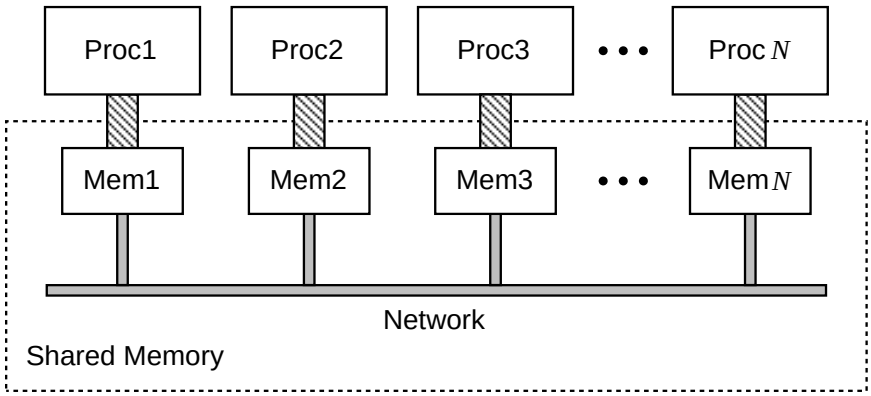

Figure 1 Distributed Shared Memory

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.

We focus on the primitives for synchronization. Shared memory can give rise to *data races.* Data races happen when accesses to shared variables by different processes are interleaved in a way that the programmer did not intend. For instance, suppose one process writes several fields of a shared record, while another one tries to read those same fields. It is quite possible that the second process reads some of the newly written fields but also reads some of the old values before they are overwritten by the first process. This situation is often undesirable. To avoid it, shared-memory parallel programs contain *synchronization.* In this particular case, a *lock* would be used to allow each of the processes exclusive access to the record.

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. 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). We are well aware of the overly simplistic nature of these example codes. They are included here for demonstration purposes only. Real applications are discussed in Section 7.

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 kept 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 by process 0 is visible to all processes before they 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. `Tmk_barrier(2)` prevents any processor from starting the next iteration before all of the `grid` values computed in the current iteration are installed.

The Traveling Salesman Problem (TSP) finds the shortest path 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. Partial tours are expanded one city at a time. If the length of a partial tour plus a lower bound on the remaining portion of the path is longer than the current shortest tour, the partial tour is not explored further, because it cannot lead to a shorter tour than the current minimum length tour.

The program keeps a queue of partial tours, with the most promising one at the head. It keeps adding promising partial tours to the queue until a path which is longer than a threshold is found

```
/* the maximum number of parallel processes supported by TreadMarks */
#define TMK_NPROCS
/* the actual number of parallel processes after Tmk_startup */
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(argc, argv) int     argc; char  **argv;
/* Terminate the calling process.  Other processes are unaffected. */
void    Tmk_exit(status) int     status;
/* Block the calling process until every other process arrives at the  barrier. */
void    Tmk_barrier(id) unsigned id;
/* Block the calling process until it acquires the specified lock. */
void    Tmk_lock_acquire(id) unsigned id;
/* Release the specified lock. */
void    Tmk_lock_release(id) unsigned id;
/* Allocate the specified number of bytes of shared memory */
char   *Tmk_malloc(size) unsigned size;
/* Free shared memory allocated by Tmk_malloc. */
void    Tmk_free(ptr) char   *ptr;
```

Figure 2 TreadMarks C Interface

```
    #define   M     1024
    #define   N     1024
    float     **grid;
    float     scratch[M][N];

    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 The TreadMarks Jacobi Program

at the head of the queue. Then, the program removes it from the queue and tries all permutations of the remaining cities. Finally, it compares the shortest tour including this path with the current shortest tour, and updates the current shortest tour if necessary. The queue and the length of the current shortest tour are shared, and access to them is guarded by locks.

3 Implementation Challenges

The provision of memory *consistency* is at the heart of a DSM system: the DSM software must move data among the processors in a manner that provides the illusion of globally shared memory. In Li's original IVY system [13], 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

```
queue_type *Queue;
int        *Shortest_length;
int        queue_lock, tour_lock;
main()
{
     Tmk_startup();
     queue_lock = 0;
     tour_lock = 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);

     Tmk_lock_acquire(queue_lock);
         Exit if queue is empty;
         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);

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

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

Figure 4 The TreadMarks TSP Program

and restarts the process. For example, Figure 5 shows the 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, while with write faults, all existing copies are invalidated and the writer retains the sole copy.

Although simple and elegant, the IVY implementation of DSM 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 must be kept low. We illustrate some of the communications problems in IVY using the examples of Figure 3 and 4.

The first problem relates to the consistency model enforced by IVY, commonly referred to as *sequential consistency* [9]. Sequential consistency requires, roughly speaking, that writes to shared memory become visible "immediately" to other processors. This is why IVY sends out invalidations before it proceeds with a write to shared memory. In many circumstances sequential consistency provides an overly strong guarantee. Consider for example the updates to the best tour and its length in TSP in Figure 4. In IVY, these two shared memory updates would cause invalidations to be sent to all other processors that cache the pages containing these variables. However, since these variables are 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

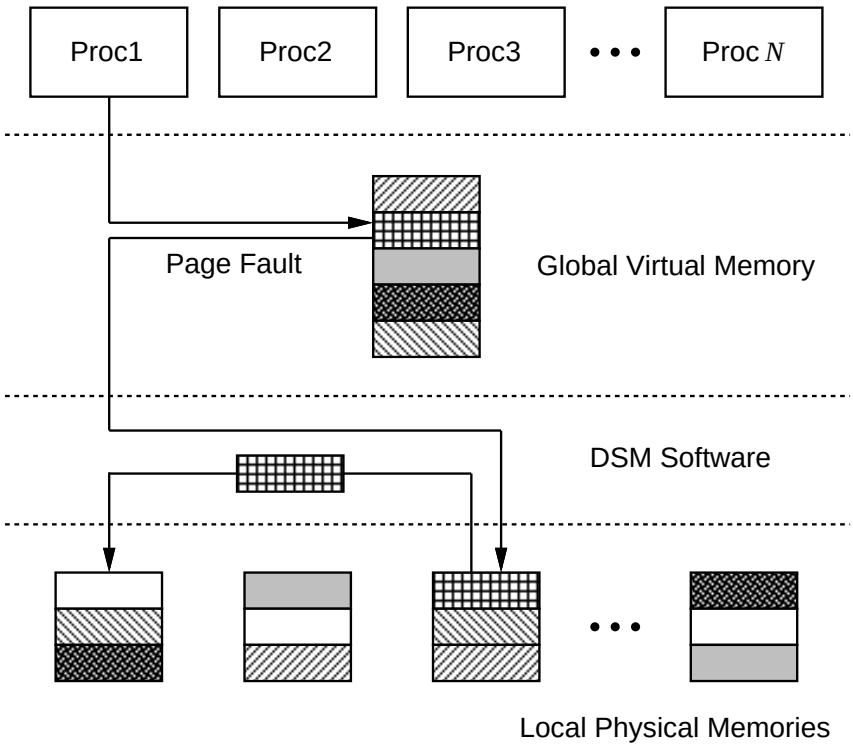

Figure 5 Operation of the IVY DSM System

or more unrelated data objects are located in the same page and are written concurrently by separate processors, causing the page to *ping-pong* back and forth between the processors. Since the consistency units are large (virtual memory pages), false sharing is a potentially serious problem. Figure 6 demonstrates a possible page layout for the `grid` array in Jacobi. As both processors update their portion of the `grid` array, they are writing concurrently to the same page, and cause the page to travel back and forth many times.

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

Release consistency [5] is a relaxed memory consistency model. It does not guarantee consistency at all times; the propagation of updates to shared memory may be delayed for some amount of time. By doing so, a release consistency implementation can coalesce updates into a single message, thereby reducing the number of messages. However, by enforcing consistency at certain synchronization operations, the potential inconsistencies are hidden from the programmer, at least

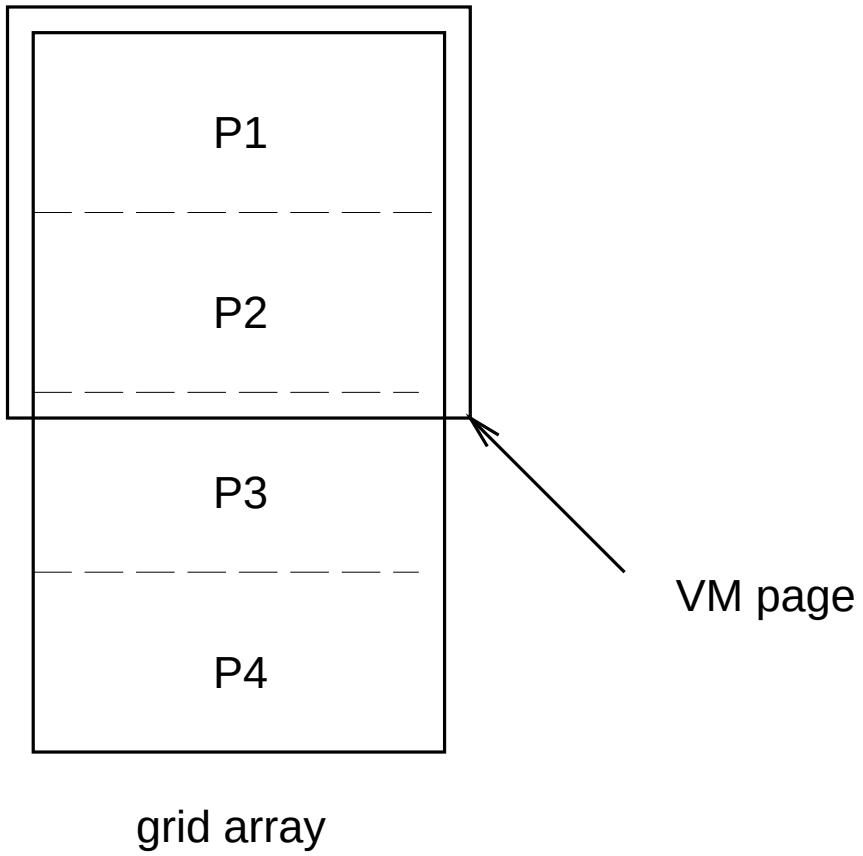

Figure 6 False Sharing Example in Jacobi

for properly synchronized programs. The discussion here is kept at an intuitive level. For a more detailed explanation, we refer the reader to the paper by Gharachorloo et al. [5].

The intuition underlying release consistency is as follows. Parallel programs should not have data races, because they lead to nondeterministic results. Thus, sufficient synchronization must be present to prevent data races. More specifically, synchronization must be present between two conflicting accesses to shared memory (Two accesses are conflicting if they refer to the same memory location and at least one of them is a write). Since this synchronization is present, the DSM system can delay sending the updates from one process to another until there is such a synchronization event, because the second process will not access the data until the synchronization operation has been executed. As a result, for programs that do not have data races, sequential consistency and release consistency lead to the same results.

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 installed in the `grid` array. This barrier needs to be there for correctness, regardless of the memory model. However, its presence allows us to postpone the propagation of the new values of the `grid` array elements until the barrier is lowered.

In TSP, the task 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. Since only the process that is holding the lock is allowed to access the data structure, there is no need to immediately propagate the shared memory updates to the other processors. These other processors will need to acquire the lock first. It therefore suffices to propagate the updates 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 looking at certain memory locations until the synchronization operation completes. From that it follows that it is not necessary to propagate the values of those memory operations until the synchronization operation completes. In contrast, sequential consistency updates each remote memory location immediately. As a result, release consistency requires far fewer messages and causes far less delay in waiting for messages to arrive.

The programmer need not be too concerned with this. If the program does not have data races, it will behave as if it were executing on an ordinary shared-memory system, *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 Lazy Release Consistency

TreadMarks uses the lazy release consistency algorithm [6] 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] which enforced consistency at the time of a release. Lazy release consistency causes fewer messages to be sent. 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 network 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.

5 Multiple-Writer Protocols

Most hardware cache and DSM systems 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 writers to simultaneously modify the same page, with consistency traffic deferred until a later time, in particular until synchronization occurs.

TreadMarks uses the virtual memory hardware to detect accesses and modifications to shared memory pages. Figure 7 shows how protection faults are used to create diffs. Valid pages are initially write-protected. When a write occurs, TreadMarks creates a copy of the virtual memory page, or a *twin*, and saves it in system space. When the modifications need to be sent out to another processor, the current copy of the page is compared with the twin on a word-by-word basis and the bytes that vary are saved into the diff data structure. Once the diff has been created, the twin is discarded. 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, but they can also significantly reduce overall bandwidth requirements because diffs are typically much smaller than a page.

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, and SGI platforms and on Ethernet and ATM networks. In this section, we briefly describe how communication and memory management by TreadMarks are implemented.

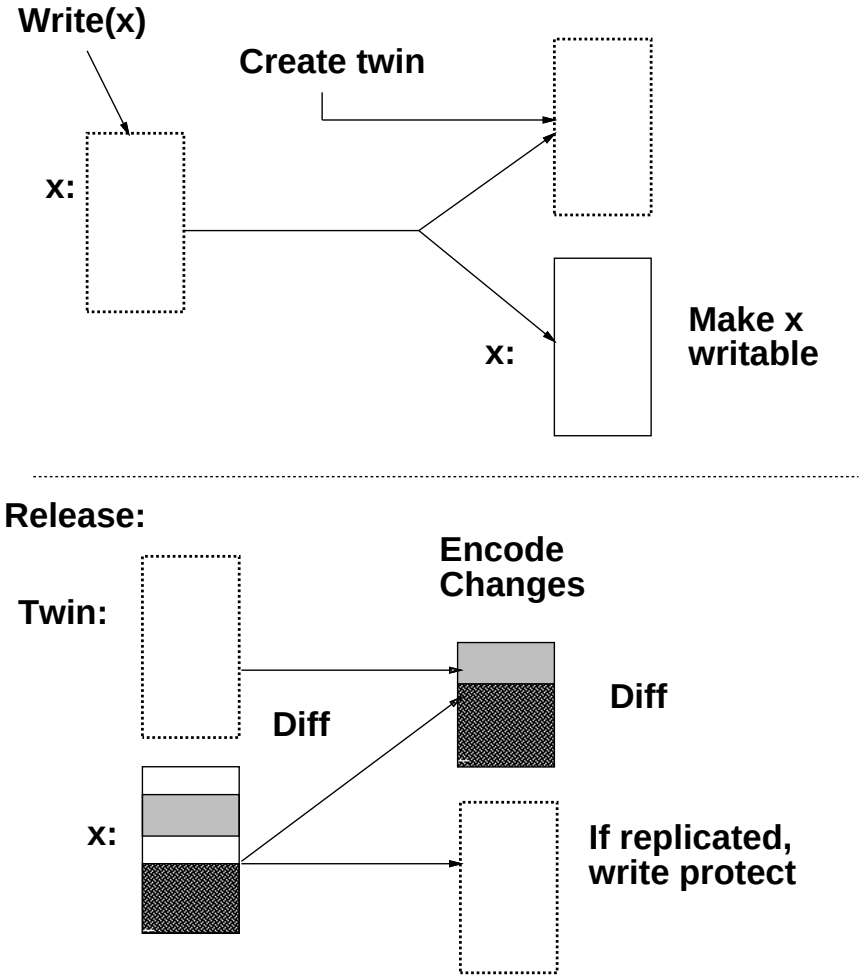

Figure 7 DiffCreation

TreadMarks implements intermachine communication using the Berkeley sockets interface. Depending on the underlying networking hardware, for example, Ethernet or ATM, TreadMarks uses either UDP/IP or AAL3/4 as the message transport protocol. By default, TreadMarks uses UDP/IP unless the machines are connected by an ATM LAN. AAL3/4 is a connection-oriented, best-efforts delivery protocol specified by the ATM standard. Since neither UDP/IP nor AAL3/4 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 response message arrives. 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. Since AAL3/4 is a connection-oriented protocol, there is a socket corresponding to each of the other machines. To determine which socket holds the incoming request, the handler performs a `select` system call. 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. If the local copy is read-only, the handler allocates a page from the pool of free pages and performs a `bcopy` to create a *twin*. Finally, the handler upgrades the access rights to the original page and returns. If the local page is invalid, the handler executes a procedure that obtains the essential updates to shared memory from the minimal set of remote machines.

7 Applications

A number of applications have been implemented using TreadMarks, and the performance of some benchmarks has been reported earlier [7]. 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.

7.1 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, and Figure 9 shows a simple two-dimensional instance.

The TreadMarks code to solve the MIP problem was implemented by Lee et al. [11]. The code 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 LP problems, one with the added constraint that $x_i = \lfloor x_i \rfloor$ and another with the added constraint that $x_i = \lceil x_i \rceil$ (see Figure 10). Over time, the algorithm generates a tree of such

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

Figure 8 The Mixed Integer Programming (MIP) Problem

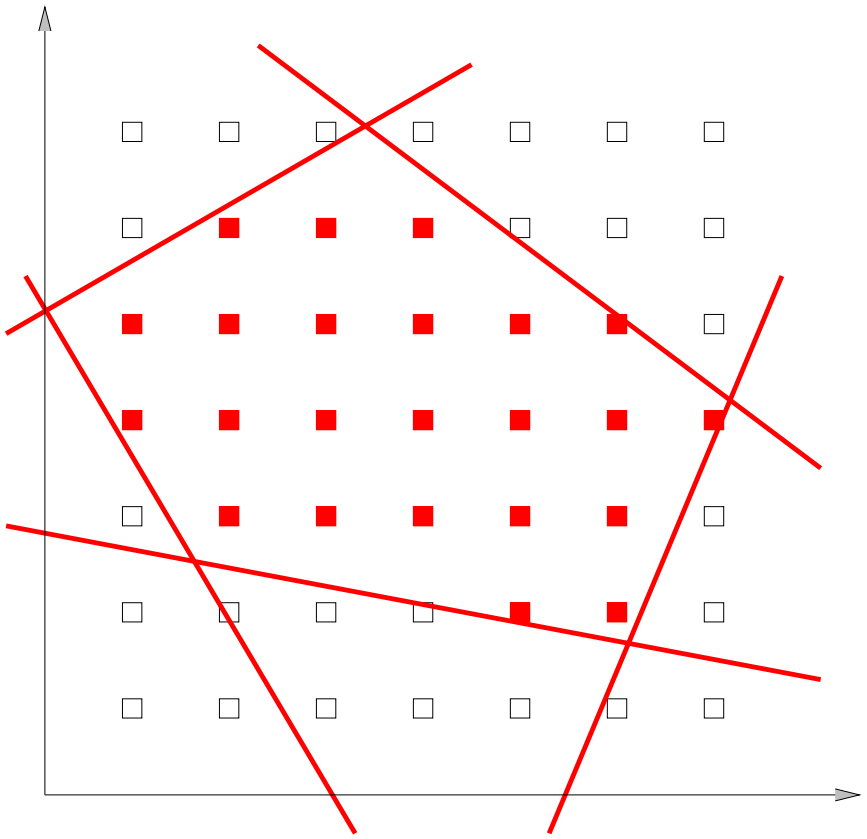

Figure 9 A Two-Dimensional Instance of MIP

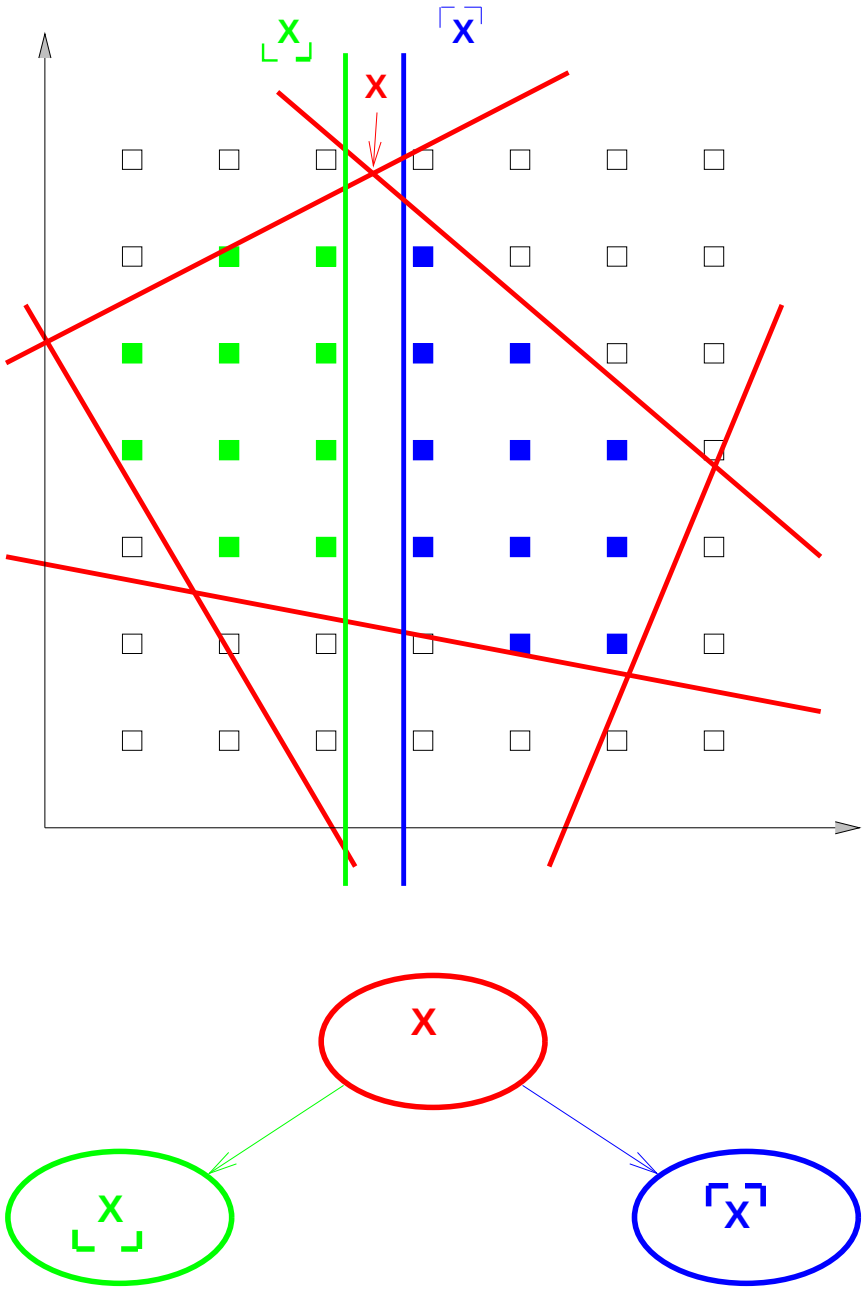

Figure 10 Branch-and-bound solution to the MIP problem

branches. As soon as a solution is found, this solution establishes a *bound* on the solution. 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 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 11 shows the speedups obtained on a network of 8 DecStation-5000/240 machines running Ultrix 4.3 and connected by a 100Mbps Fore ATM switch, for those problems whose sequential running time is 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 [2]. The problem took roughly 30 CPU days on an 8-processor IBM SP-1 and also exhibited near-linear speedup.

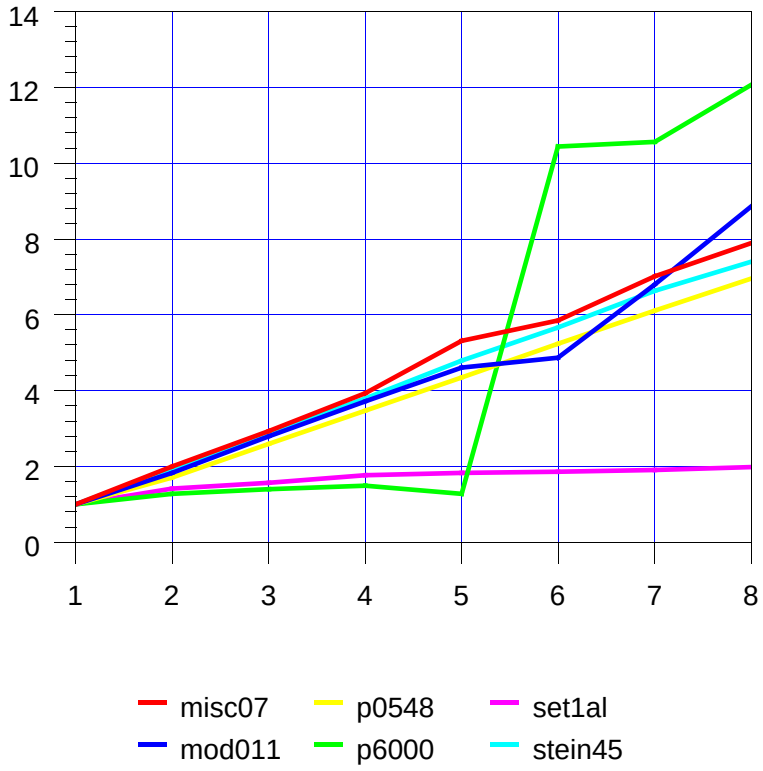

Figure 11 Results from the MIPLIB Library

7.2 Genetic Linkage

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.

In the classical Mendelian theory of inheritance, the child's chromosomes receive one strand of each of the parent's chromosomes. When taking *recombination* into account, the situation becomes a bit different. If a recombination occurs, the child's chromosome strand will contain a piece of each of the strands of the parent's chromosome (see Figure 12). 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 [4] is a parallelized version of a widely used genetic linkage analysis program, which is part

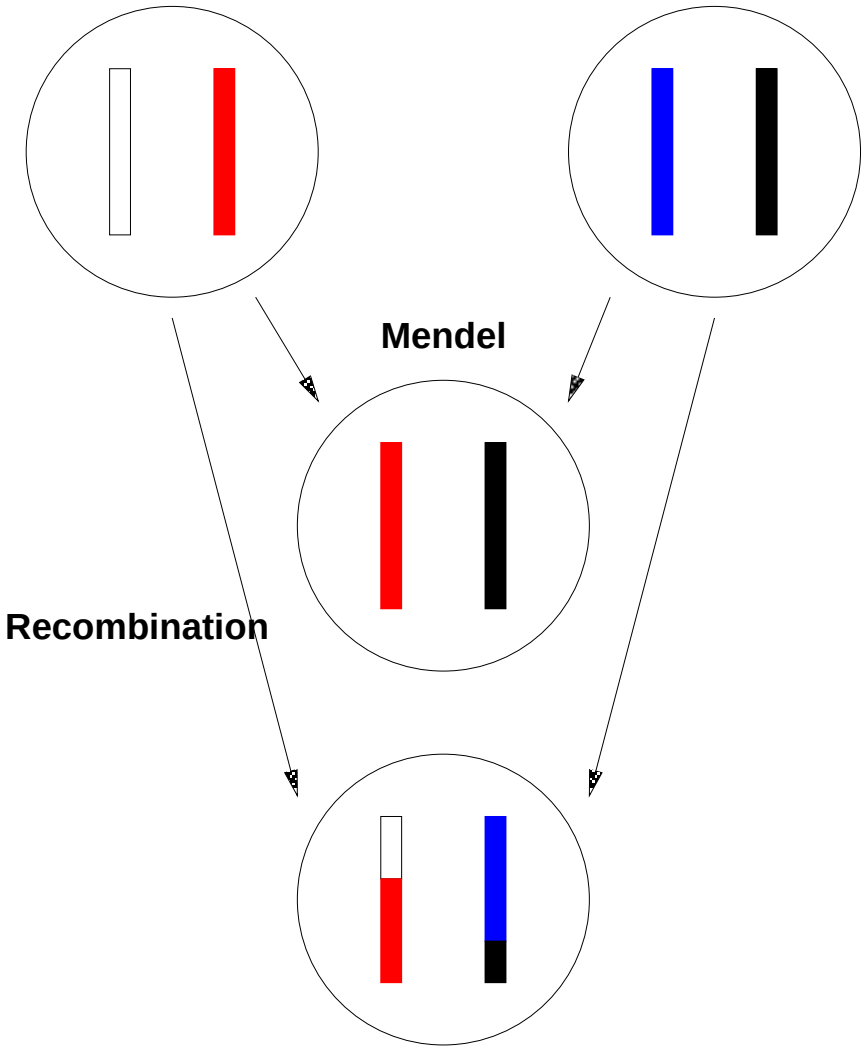

Figure 12 DNA Recombination

of the LINKAGE [10] package. 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, 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). Also, parallel evaluation for different θ value is not possible because the optimization procedure moves sequentially from one θ value to the next.

Figure 13 presents speedups obtained for various data sets using ILINK. The data sets originate from real disease gene location studies. For the data sets with a long running time, good speedups are achieved. For the smallest data set, BAD, speedup is much less because the communication-to-computation ratio becomes larger.

8 Related Work

Our goal in this section is not to provide an extensive survey of parallel programming research, but instead to illustrate alternative approaches with one distinctive example, and compare the resulting system to TreadMarks.

Message Passing (PVM). Currently, message passing is the prevailing programming paradigm for distributed memory systems. Portable Virtual Machine (PVM) [15] is a popular software message passing package. It allows a heterogeneous network of computers to appear as a single concurrent computational engine. 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.

Implicit Parallelism (HPF). Implicit parallelism, as in HPF [8], relies on user-provided data distributions which are then used by the compiler to generate message passing code. This approach is suitable for data-parallel programs, such as Jacobi. Programs exhibiting dynamic parallelism, such as TSP, ILINK, or MIP are not easily expressed in the HPF framework.

Object-Oriented Parallel Computing (Orca). Rather than providing the programmer with a shared memory space, Orca [16] and other object-oriented parallel computing systems support a shared space of object, each of which is 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. On the downside, the

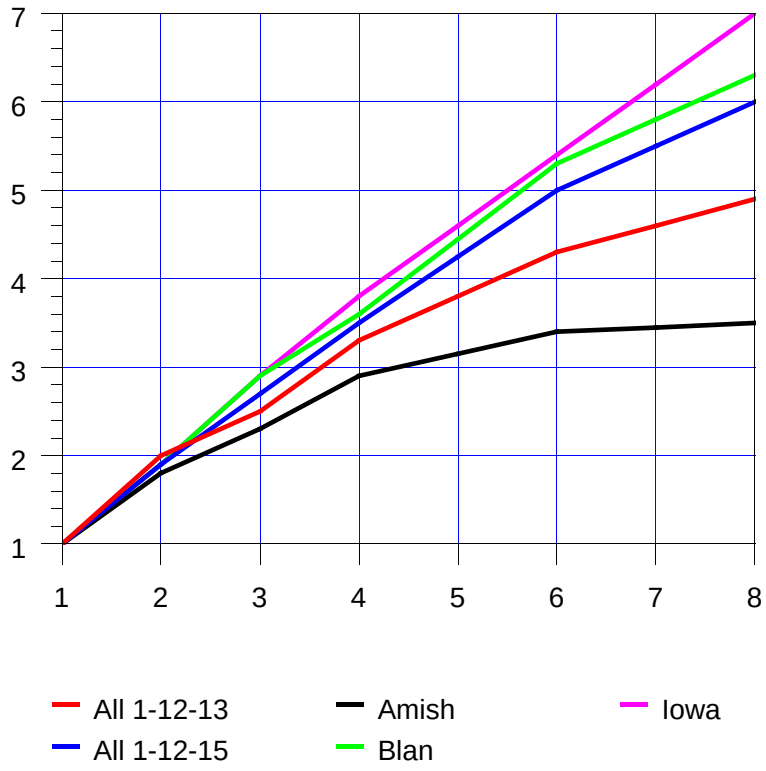

Figure 13 ILINK Results

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.

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., [12]). 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.

Entry Consistency (Midway). Entry consistency is another relaxed memory model [1]. It requires all shared data to be associated with a synchronization object. When a synchronization object is acquired, only the modified data associated with that synchronization object is transferred, leading to further reductions in the amount of communication. However, the entry consistency memory model is weaker than release consistency, potentially making programming somewhat harder.

9 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. Major 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 such as prefetching and the use of performance monitoring tools to eliminate unnecessary synchronization.

References

[1] 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.

[2] D. Bienstock and O. Gumluk. Computational experience with a difficult mixed-integer multicommodity flow problem. To appear in Mathematical Programming, 1994.

[3] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[4] S. Dwarkadas, A.A. Schäffer, R.W. Cottingham Jr., A.L. Cox, P. Keleher, and W. Zwaenepoel. Parallelization of general linkage analysis problems. *Human Heredity*, 44:127–141, 1994.

[5] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[6] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[7] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

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

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

[10] G. M. Lathrop, J. M. Lalouel, C. Julier, and J. Ott. Strategies for multilocus linkage analysis in humans. *Proc. Natl. Acad. Sci. USA*, 81:3443–3446, June 1984.

[11] E. Lee, R. Bixby, W. Cook, and A.L. Cox. Parallelism in mixed integer programming. Submitted for publication, 1994.

[12] 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.

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

[14] M. Litzkow, M. Livny, and M. Mutka. Condor — a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, June 1988.

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

[16] A.S. Tanenbaum, M.F. Kaashoek, and H.E. Bal. Parallel programming using shared objects and broadcasting. *IEEE Computer*, 25(8):10–20, August 1992.

[17] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.