Message Passing Versus Distributed Shared Memory on Networks of Workstations

Honghui Lu
Department of Electrical and Computer Engineering
Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel
Department of Computer Science
Rice University
Houston, TX 77005
e-mail: {hhl, sandhya, alc, willy}@cs.rice.edu

Abstract

We compare two paradigms for parallel programming on networks of workstations: message passing and distributed shared memory. We present results for nine applications that were implemented using both paradigms.

The message passing programs are executed with the Parallel Virtual Machine (PVM) library and the shared memory programs are executed using TreadMarks. The programs are Water and Barnes-Hut from the SPLASH benchmark suite; 3-D FFT, Integer Sort (IS) and Embarrassingly Parallel (EP) from the NAS benchmarks; ILINK, a widely used genetic linkage analysis program; and Successive Over-Relaxation (SOR), Traveling Salesman (TSP), and Quicksort (QSORT). Two different input data sets were used for Water (Water-288 and Water-1728), IS (IS-Small and IS-Large), and SOR (SOR-Zero and SOR-NonZero). Our execution environment is a set of eight HP735 workstations connected by a 100Mbits per second FDDI network.

For Water-1728, EP, ILINK, SOR-Zero, and SOR-NonZero, the performance of TreadMarks is within 10% of PVM. For IS-Small, Water-288, Barnes-Hut, 3-D FFT, TSP, and QSORT, differences are on the order of 10% to 30%. Finally, for IS-Large, PVM performs two times better than TreadMarks.

More messages and more data are sent in TreadMarks, explaining the performance differences. This extra communication is caused by 1) the separation of synchronization and data transfer, 2) extra messages to request updates for data by the invalidate protocol used in TreadMarks, 3) false sharing, and 4) *diff accumulation* for migratory data in TreadMarks.

1 Introduction

Parallel computing on networks of workstations has been gaining more attention in recent years. Because workstation clusters use "off the shelf" products, they are cheaper than supercomputers. Furthermore, high-

This research was supported in part by NSF NYI Award CCR-9457770, NSF CISE postdoctoral fellowship Award CDA-9310073, NSF Grants CCR-9116343 and BIR-9408503, and by the Texas Advanced Technology Program under Grant 003604012.

speed general-purpose networks and very powerful workstation processors are narrowing the performance gap between workstation clusters and supercomputers.

Processors in workstation clusters do not share physical memory, so all interprocessor communication between processors must be performed by sending messages over the network. Currently, the prevailing programming model for parallel computing on networks of workstations is message passing, using libraries such as PVM [9], TCGMSG [11] and Express [18]. A message passing standard MPI [17] has also been developed. With the message passing paradigm, the distributed nature of the memory system is fully exposed to the application programmer. The programmer needs to keep in mind where the data is, decide *when* to communicate with other processors, *whom* to communicate with, and *what* to communicate, making it hard to program in message passing, especially for applications with complex data structures.

Software distributed shared memory (DSM) systems (e.g., [3, 5, 14, 16]) provide a shared memory abstraction on top of the native message passing facilities. An application can be written as if it were executing on a shared memory multiprocessor, accessing shared data with ordinary read and write operations. The chore of message passing is left to the underlying DSM system. While it is easier to program this way, DSM systems tend to generate more communication and therefore tend to be less efficient than message passing systems. Under the message passing paradigm, communication is handled entirely by the programmer, who has complete knowledge of the data usage pattern. In contrast, the DSM system has little knowledge of the application program, and therefore must be conservative in determining what to communicate. Since sending messages between workstations is expensive, this extra communication can cause serious performance degradation.

Much work has been done in the past decade to improve the performance of DSM systems. In this paper, we compare a state-of-the-art DSM system, TreadMarks, with the most commonly used message passing system, PVM. Our goals are to assess the differences in programmability and performance between DSM and message passing systems and to precisely determine the remaining causes of the lower performance of DSM systems.

We ported nine parallel programs to both TreadMarks and PVM: Water and Barnes-Hut from the SPLASH benchmark suite [20]; 3-D FFT, Integer Sort (IS), and Embarrassingly Parallel (EP) from the NAS benchmarks [2]; ILINK, a widely used genetic linkage analysis program [8]; and Successive Over-Relaxation (SOR), Traveling Salesman Problem (TSP), and Quicksort (QSORT). Two different input sets were used for Water (Water-288 and Water-1728), IS (IS-Small and IS-Large), and SOR (SOR-Zero and SOR-NonZero). We ran these programs on eight HP735 workstations connected by a 100Mbits per second FDDI network.

In terms of programmability, since most of our test programs are simple, it was not difficult to port them to PVM. However, for two of the programs, namely 3-D FFT and ILINK, the message passing versions were significantly harder to develop than the DSM versions.

For Water-1728, EP, ILINK, SOR-Zero, and SOR-NonZero, the performance of TreadMarks is within 10% of PVM. For IS-Small, Water-288, Barnes-Hut, 3-D FFT, TSP, and QSORT, differences are on the order of 10% to 30%. Finally, for IS-Large, PVM performs two times better than TreadMarks.

More messages and more data are sent in TreadMarks, explaining the performance differences. This extra communication is caused by 1) the separation of synchronization and data transfer, 2) extra messages to request updates for data in the invalidate protocol used in TreadMarks, 3) false sharing, and 4) *diff accumulation* for migratory data in TreadMarks. We are currently trying to address these deficiencies through the integration of compiler support in TreadMarks.

The rest of this paper is organized as follows. In Section 2 we introduce the user interfaces and implementations of PVM and TreadMarks. Section 3 presents the application programs and their results. Section 4 concludes the paper.

2 PVM Versus TreadMarks

2.1 PVM

PVM [9], standing for Parallel Virtual Machine, is a message passing system originally developed at Oak Ridge National Laboratory. Although other message passing systems such as TCGMSG [11], provide higher bandwidth than PVM, we chose PVM because of its popularity. We use PVM version 3.2.6 in our experiments.

2.1.1 PVM Interface

With PVM, the user data must be *packed* into a send buffer before being dispatched. The received message is first stored in a receive buffer, and must be *unpacked* into the application data structure. The application program calls different routines to pack or unpack data with different types. All these routines have the same syntax, which specifies the beginning of the user data structure, the total number of data items to be packed or unpacked, and the stride. The unpack calls should match the corresponding pack calls in type and number of items.

PVM provides the user with nonblocking *sends*, including primitives to send a message to a single destination, to multicast to multiple destinations, or to broadcast to all destinations. The send dispatches the contents of the send buffer to its destination and returns immediately.

Both blocking and nonblocking *receives* are provided by PVM. A receive provides a receive buffer for an incoming message. The blocking receive waits until an expected message has arrived. At that time, it returns a pointer to the receive buffer. The nonblocking receive returns immediately. If the expected message is present, it returns the pointer to the receive buffer, as with the blocking receive. Otherwise, the nonblocking receive returns a null pointer. Nonblocking receive can be called multiple times to check for the presence of the same message, while performing other work between calls. When there is no more useful work to do, the blocking receive can be called for the same message.

2.1.2 PVM Implementation

PVM consists of two parts: a daemon process on each host and a set of library routines. The daemons connect with each other using UDP, and a user process connects with its local daemon using TCP. The usual way for two user processes on different hosts to communicate with each other is via their local daemons. They can, however, set up a direct TCP connection between each other in order to reduce overhead. We use a direct connection between the user processors in our experiments, because it results in better performance.

Because PVM is designed to work on a set of heterogeneous machines, it provides conversion to and from an external data representation (XDR). This conversion is avoided if all the machines used are identical.

2.2 TreadMarks

TreadMarks [14] is a software DSM system built at Rice University. It is an efficient user-level DSM system that runs on commonly available Unix systems. We use TreadMarks version 0.9.4 in our experiments.

2.2.1 TreadMarks Interface

TreadMarks provides primitives similar to those used in hardware shared memory machines. Application processes synchronize via two primitives: barriers and *mutex* locks. The routine `Tmk_barrier(i)` stalls the calling process until all processes in the system have arrived at the same barrier. Barrier indices `i` are integers in a certain range. Locks are used to control access to critical sections. The routine `Tmk_lock_acquire(i)` acquires a lock for the calling processor, and the routine `Tmk_lock_release(i)` releases it. No processor can acquire a lock if another processor is holding it. The integer `i` is a lock index assigned by the programmer. Shared memory must be allocated dynamically by calling `Tmk_malloc` or `Tmk_sbrk`. They have the same syntax as conventional memory allocation calls. With TreadMarks, it is imperative to use explicit synchronization, as data is moved from processor to processor only in response to synchronization calls (see Section 2.2.2).

2.2.2 TreadMarks Implementation

TreadMarks uses a *lazy invalidate* [14] version of *release consistency* (RC) [10] and a multiple-writer protocol [5] to reduce the amount of communication involved in implementing the shared memory abstraction. The virtual memory hardware is used to detect accesses to shared memory.

RC is a relaxed memory consistency model. In RC, *ordinary* shared memory accesses are distinguished from *synchronization* accesses, with the latter category divided into *acquire* and *release* accesses. RC requires ordinary shared memory updates by a processor p to become visible to another processor q only when a subsequent release by p becomes visible to q via some chain of synchronization events. In practice, this model allows a processor to buffer multiple writes to shared data in its local memory until a synchronization point is reached. In TreadMarks, `Tmk_lock_acquire(i)` is modeled as an acquire, and `Tmk_lock_release(i)` is modeled as a release. `Tmk_barrier(i)` is modeled as a release followed by an acquire, where each processor performs a release at barrier arrival, and an acquire at barrier departure.

With the multiple-writer protocol, two or more processors can simultaneously modify their own copy of a shared page. Their modifications are merged at the next synchronization operation in accordance with the definition of RC, thereby reducing the effect of false sharing. The merge is accomplished through the use of *diffs*. A diff is a runlength encoding of the modifications made to a page, generated by comparing the page to a copy saved prior to the modifications.

TreadMarks implements a *lazy invalidate* version of RC [13]. A *lazy* implementation delays the propagation of consistency information until the time of an acquire. Furthermore, the releaser notifies the acquirer of which pages have been modified, causing the acquirer to *invalidate* its local copies of these pages. A processor incurs a page fault on the first access to an invalidated page, and gets diffs for that page from previous releasers.

To implement lazy RC, the execution of each processor is divided into *intervals*. A new interval begins every time a processor synchronizes. Intervals on different processors are partially ordered: (i) intervals on a single processor are totally ordered by program order, (ii) an interval on processor p *precedes* an interval on processor q if the interval of q begins with the acquire corresponding to the release that concluded the interval of p, and (iii) an interval precedes another interval by transitive closure. This partial order is known as *hb1* [1]. Vector *timestamps* are used to represent the partial order.

When a processor executes an acquire, it sends its current timestamp in the acquire message. The previous releaser then piggybacks on its response the set of *write notices* that have timestamps greater than the timestamp in the acquire message. These write notices describe the shared memory modifications that precede the acquire according to the partial order. The acquiring processor then invalidates the pages for which there are incoming write notices.

On an access fault, a page is brought up-to-date by fetching all the missing diffs and applying them to the page in increasing timestamp order. All write notices without corresponding diffs are examined. It is usually unnecessary to send diff requests to all the processors who have modified the page, because if a processor has modified a page during an interval, then it must have all the diffs of all intervals that precede it, including those from other processors. TreadMarks then sends diff requests to the subset of processors for which their most recent interval is not preceded by the most recent interval of another processor.

Each lock has a statically assigned manager. The manager records which processor has most recently requested the lock. All lock acquire requests are directed to the manager, and, if necessary, forwarded to the processor that last requested the lock. A lock release does not cause any communication. Barriers have a centralized manager. The number of messages sent in a barrier is $2 \times (n-1)$, where n is the number of processors.

3 Results

3.1 Experimental Testbed

The testbed we used to evaluate the two systems is an 8-node cluster of HP9000-735/125 workstations, each with a 125Mhz PA-RISC7100 processor and 96 megabytes of main memory. The machines have a 4096-byte page size and are connected by a 100Mbps FDDI ring.

In TreadMarks, the user processes communicate with each other using UDP. In PVM, processes set up direct TCP connections with each other. Since all the machines are identical, data conversion to and from external data representation is disabled. Both UDP and TCP are built on top of IP, with UDP being connectionless and TCP being connection oriented. TCP is a reliable protocol while UDP does not ensure reliable delivery. TreadMarks uses light-weight, operation-specific, user-level protocols on top of UDP to ensure reliable delivery.

3.2 Overview

We ported nine parallel programs to both TreadMarks and PVM: Water and Barnes-Hut from the SPLASH benchmark suite [20]; 3-D FFT, IS, and EP from the NAS benchmarks [2]; ILINK, a widely used genetic linkage analysis program [8]; and SOR, TSP, and QSORT. We ran three of the nine programs using two different input sets: Water with 288 and 1728 molecules, IS with a bucket size of 2^{10} and 2^{15}, and SOR with the internal elements of the matrix initialized to either zero or nonzero values. The execution times for the sequential programs, without any calls to PVM or TreadMarks, are shown in Table 1. This table also shows the problem sizes used for each application. Figures 1 to 12 show the speedup curves for each of the applications. The speedup is computed relative to the sequential program execution times given in Table 1. The amount of data and the number of messages sent during the 8-processor execution are shown in Table 2. In the PVM versions, we counted the number of user-level messages and the amount of user data sent in each run. In TreadMarks, we counted the total number of UDP messages, and the total amount of data communicated.

Program	Problem Size	Time(sec.)
EP	2^{28}	2391
SOR-Zero	1024×3072, 50 iterations	79
SOR-Nonzero	1024×3072, 50 iterations	68
IS-Small	$N = 2^{23}$, $B_{max} = 2^{10}$,10 iterations	38
IS-Large	$N = 2^{23}$, $B_{max} = 2^{15}$,10 iterations	42
TSP	19 cities	95
QSORT	512k integers	53
Water-288	288 molecules, 5 iterations	12
Water-1728	1728 molecules, 5 iterations	435
Barnes-Hut	4096 bodies	17
3-D FFT	$64 \times 64 \times 64$, 6 iterations	41
ILINK	CLP	1473

Table 1 Sequential Time of Applications

Program	TreadMarks		PVM	
	Messages	Kilobytes	Messages	Kilobytes
EP	86	33	7	0.3
SOR-Zero	7014	509	1414	8607
SOR-Nonzero	7014	8853	1414	8607
IS-Small	872	2582	140	573
IS-Large	9918	73591	140	18350
TSP	18601	5780	1480	38
QSORT	33742	57083	3129	16000
Water-288	5028	4945	620	1520
Water-1728	8511	21170	620	9123
Barnes-Hut	62350	18277	280	12704
3-D FFT	14686	28732	1834	25690
ILINK	255047	119969	6615	47583

Table 2 Messages and Data at 8 Processors

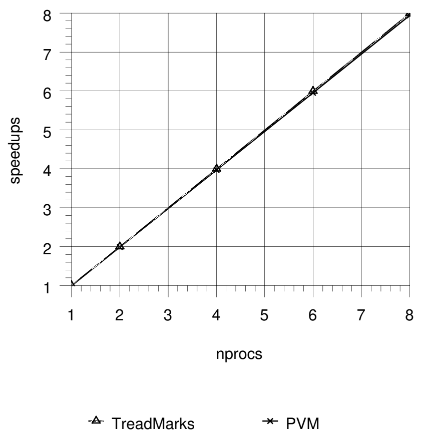

Figure 1 EP

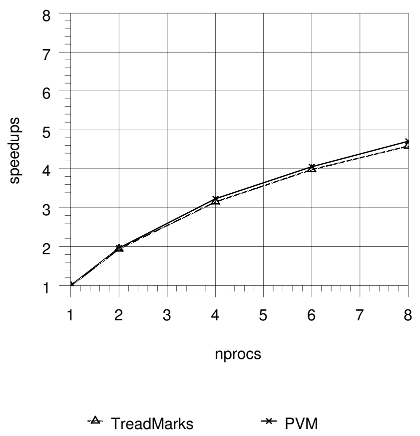

Figure 2 SOR-Zero

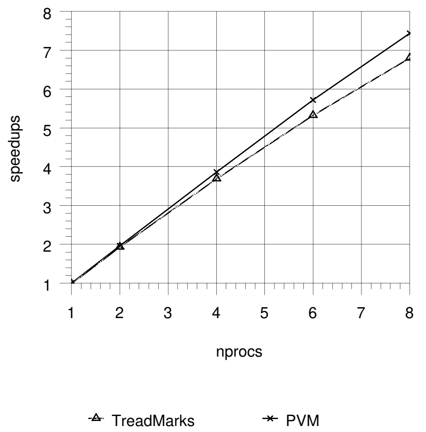

Figure 3 SOR-Nonzero

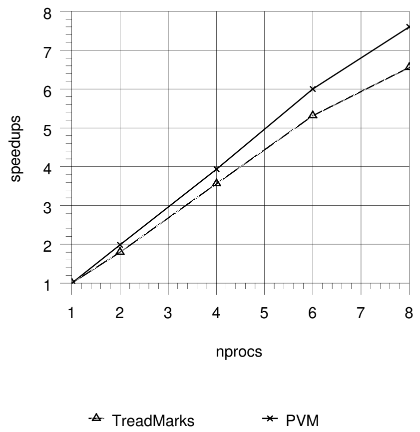

Figure 4 IS-Small

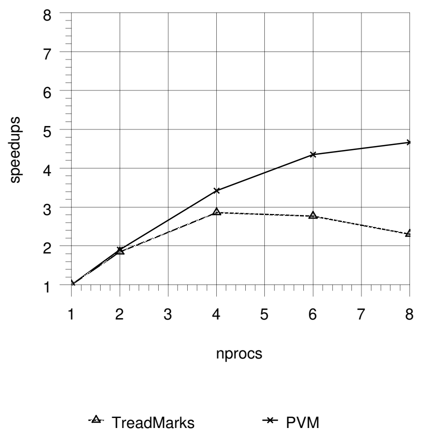

Figure 5 IS-Large

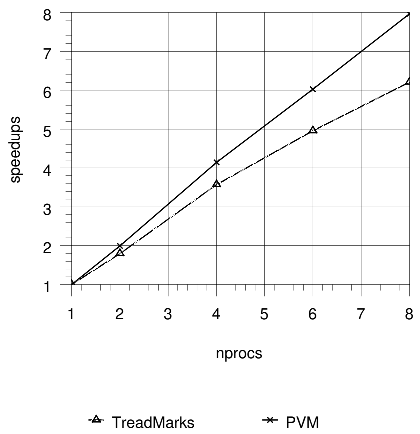

Figure 6 TSP

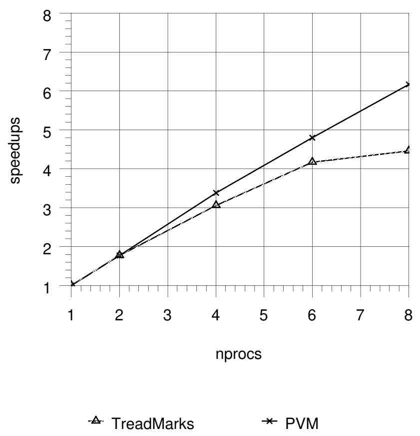

Figure 7 QSORT

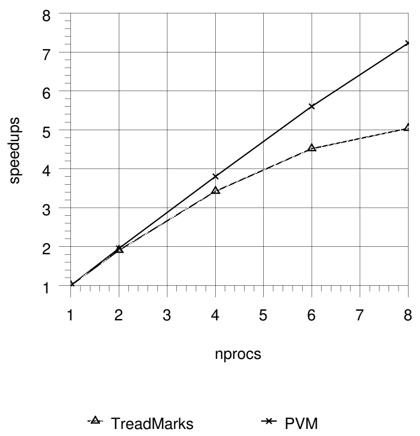

Figure 8 Water-288

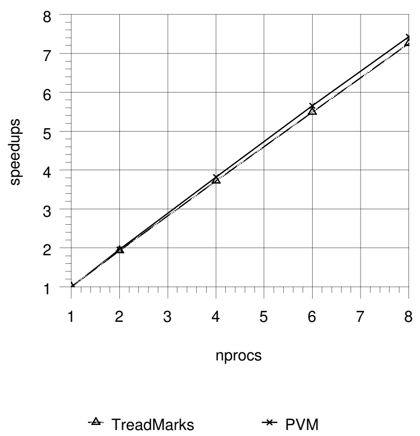

Figure 9 Water-1728

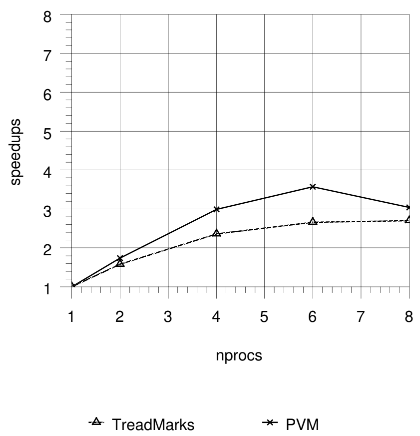

Figure 10 Barnes-Hut

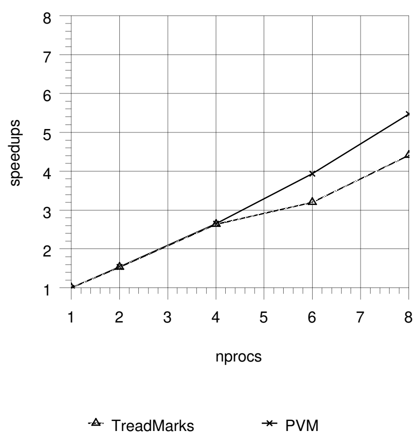

Figure 11 3-D FFT

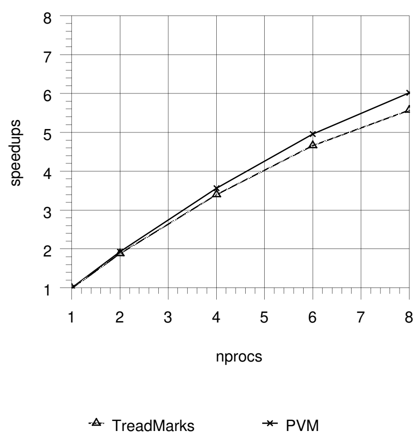

Figure 12 ILINK

3.3 EP

The Embarrassingly Parallel program comes from the NAS benchmark suite [2]. EP generates pairs of Gaussian random deviates and tabulates the number of pairs in successive square annuli. In the parallel version, the only communication is summing up a ten-integer list at the end of the program. In TreadMarks, updates to the shared list are protected by a lock. In PVM, processor 0 receives the lists from each processor and sums them up.

In our test, we solved the class A problem in the NAS benchmarks, in which 2^{28} pairs of random numbers are generated. The results are shown in Figure 1. The sequential program runs for 2391 seconds. Both TreadMarks and PVM achieve a speedup of 7.9 using 8 processors, because compared to the overall execution time, the communication overhead is negligible.

3.4 Red-Black SOR

Red-Black Successive Over-Relaxation (SOR) is a method of solving partial differential equations. In the parallel version, the program divides the red and the black array into roughly equal size bands of rows, assigning each band to a different processor. Communication occurs across the boundary rows between bands. In the TreadMarks version, the arrays are allocated in shared memory, and processors synchronize using barriers. With PVM, each processor explicitly sends the boundary rows to its neighbors.

We ran red-black SOR on a 1024×3072 matrix of floating point numbers for 51 iterations. With this problem size each shared red or black row occupies one and a half pages. The first iteration is excluded from measurement to eliminate differences due to the fact that data is initialized in a distributed manner in the PVM version, while in TreadMarks it is done at the master process.

In the first test (SOR-Zero), the edge elements are initialized to 1, and all the other elements to 0. In the second test (SOR-Nonzero), all elements of the matrix are initialized to nonzero values, such that they all change values in each iteration.

Results from SOR-Zero are shown in Figure 2. The sequential program runs for 79 seconds. At 8 processors, the TreadMarks version and the PVM version achieve speedups of 4.58 and 4.71, respectively. The TreadMarks speedup is 97% that of PVM. Due to load imbalance, neither PVM nor TreadMarks achieves good speedup. Load imbalance occurs because floating-point computations involving zeros take longer than those involving non-zeros, causing the processors working on the middle parts of the array to take longer between iterations. Results from SOR-Nonzero are shown in Figure 3. Because the initial values are nonzero, the single processor time drops from 79 seconds to 68 seconds. At 8 processors, the speedup obtained by TreadMarks is 6.80, which is 91% of the PVM speedup of 7.44. Compared to the first test, the improved speedup is due to better load balance.

TreadMarks and PVM performance are relatively close, because of the low communication rate in SOR, and the use of lazy release consistency in TreadMarks. Although each processor repeatedly writes to the boundary pages between two barriers, diffs of the boundary pages are sent only once after each barrier, in response to diff requests from neighbors. The number of messages is 5 times higher in TreadMarks than in PVM. For n processors, PVM sends $2 \times (n-1)$ messages at the end of each iteration. TreadMarks sends $2 \times (n-1)$ messages to implement the barrier and $8 \times (n-1)$ messages to page in the diffs for the boundary rows (Each boundary row requires two diffs, one for each page). This behavior exemplifies two of the performance drawbacks of TreadMarks relative to PVM: separation of synchronization and data transfer and multiple diff requests due to the invalidate protocol. As a result of diffing in TreadMarks, much less data is sent in SOR-Zero by TreadMarks than by PVM because most of the pages remain zero.

3.5 Integer Sort

Integer Sort (IS) [2] from the NAS benchmarks requires ranking an unsorted sequence of keys using bucket sort. The parallel version of IS divides up the keys among the processors. First, each processor counts its keys and writes the result in a private array of buckets. Then, the values in the private buckets are summed up. Finally, all processors read the sum and rank their keys.

In the TreadMarks version, there is a shared array of buckets, and each processor also has a private array of buckets. After counting its keys, a processor locks the shared array of buckets, adds the values in its private array to the shared array, releases the lock, and waits at a barrier until all other processors have finished their updates. Each processor then reads the final result in the shared array of buckets and ranks its keys. In the PVM version, each processor has a bucket array in private memory. After counting their own keys, the processors form a chain, in which processor 0 sends its local array of buckets to processor 1. Processor 1 adds the values in its local array of buckets to the values in the array of buckets it receives and forwards the result to the next processor, etc. The last processor in the chain calculates the final result and broadcasts it.

We tested IS with two sets of input data. In the first test (IS-Small), we sorted 2^{23} keys ranging from 0 to 2^{10} for 10 iterations. In the second test (IS-Large), the keys range from 0 to 2^{15}. We did not try the 2^{23} keys and the 2^{20} key range as suggested for the NAS benchmarks, because the extremely low computation/communication ratio is not suitable for workstation clusters.

The speedups are shown in Figures 4 and 5. The sequential execution time for IS-Small is 38 seconds. The 8 processor speedups for PVM and TreadMarks are 7.60 and 6.56, respectively. For IS-Large, the sequential program runs for 42 seconds, and PVM and TreadMarks achieve speedups of 4.67 and 2.30, respectively.

In IS-Small, the TreadMarks version sends 4 times more data and 5 times more messages than the PVM version. The extra messages are due to separate synchronization and diff requests. Of the 782 messages sent in TreadMarks (compared to 140 in PVM), 500 are synchronization messages, and about 150 are diff requests. In IS-Large, TreadMarks sends about 70 times more messages than PVM. The shared bucket array in IS-Large contains 2^{15} integers, spread over 32 pages. Therefore, each time the shared bucket array is accessed, TreadMarks sends 32 diff requests and responses, while PVM handles the transmission of the shared array with a single message exchange.

The extra data in TreadMarks comes from a phenomenon we call *diff accumulation*. Each time a processor acquires a lock to modify the shared array of buckets, the previous values in the array are completely overwritten. In the current TreadMarks implementation, however, all the preceding diffs are sent when a lock is acquired, even though (for IS) they completely overlap each other. The same phenomenon occurs after the barrier, when every processor reads the final values in the shared bucket. At this time, each processor gets all the diffs made by the processors who modified the shared bucket array after it during this iteration. Assuming the bucket size is b and the number of processors is n, in PVM, the amount of data sent in each iteration is $2 \times (n-1) \times b$, while the amount of data sent in TreadMarks is $n \times (n-1) \times b$. Although all the diffs can be obtained from one processor, diff accumulation also results in more messages when the sum of the diff sizes exceeds the maximum size of a UDP message. Since the TreadMarks MTU is 48 kilobytes, extra messages due to diff accumulation are not a serious problem.

3.6 TSP

TSP solves the traveling salesman problem using a branch and bound algorithm. The major data structures are a pool of partially evaluated tours, a priority queue containing pointers to tours in the pool, a stack of pointers to unused tour elements in the pool, and the current shortest path. The evaluation of a partial tour is composed mainly of two procedures, `get_tour` and `recursive_solve`. The subroutine `get_tour` removes the most promising path from the priority queue. If the path contains more than a threshold number of cities, `get_tour` returns this path. Otherwise, it extends the path by one node, puts the promising paths generated by the extension back on the priority queue, and calls itself recursively. The subroutine `get_tour` returns either when the most promising path is longer than a threshold, or when the priority queue becomes empty. The procedure `recursive_solve` takes the path returned by `get_tour`, and tries all permutations of the remaining nodes recursively. It updates the shortest tour if a complete tour is found that is shorter than the current best tour.

In the TreadMarks version, all the major data structures are shared. The subroutine `get_tour` is guarded by a lock to guarantee exclusive access to the tour pool, the priority queue, and the tour stack. Updates to the shortest path are also protected by a lock. The PVM version uses a master-slave arrangement. With n processors, there are n slave processes and 1 master process. In other words, one processor runs both the master and one slave process, while the remaining processors run only a slave process. The master keeps all the major data structures in its private memory. It executes `get_tour` and keeps track of the optimal solution. The slaves execute `recursive_solve`, and send messages to the master either to request solvable tours, or to update the shortest path.

We solved a 19 city problem, with a `recursive_solve` threshold of 12. The speedups are shown in Figure 6. The sequential program runs for 95 seconds. At 8 processors, TreadMarks obtains a speedup of 6.21, which is 78% of the speedup of 7.97 obtained by PVM. At 8 processors, TreadMarks sends 11 times more messages and 150 times more data than PVM.

The performance gap comes from the difference in programming styles. In the PVM version of TSP, only the tours directly solvable by `recursive_solve` and the minimum tour are exchanged between the slaves and the master. These message exchanges take only 2 messages. In contrast, in TreadMarks, all the major data structures migrate among the processors. In `get_tour`, it takes at least 3 page faults to obtain the tour pool, priority queue, and tour stack. As for the amount of data, because of diff accumulation, on average, a processor gets $(n-1)$ diffs on each page fault, where n is the number of processors in the system. Furthermore, there is some contention for the lock protecting `get_tour`. On average, at 8 processors, each process spends 1.1 out of 15 seconds waiting at lock acquires.

3.7 QSORT

In QSORT, the quicksort algorithm is used to partition an unsorted list into sublists. When the sublist is sufficiently small, the integers are sorted using bubblesort. QSORT is parallelized using a work queue that contains descriptions of unsorted sublists, from which worker threads continuously remove the lists.

In the TreadMarks version of QSORT, the list and the work queue are shared, and accesses to the work queue are protected by a lock. Unlike TSP, in QSORT, the processor releases the task queue without subdividing the subarray it removes from the queue. If the subarray is further divided, the processor reacquires control of the task queue, and places the newly generated subarrays back on the task queue. The PVM version uses a master-slave arrangement similar to TSP, with n slaves and 1 master, where n is the number of processors. The master maintains the work queue and the slaves perform the partitioning and the sorting.

In our experiments, the array size was 512K and the bubblesort threshold was 1024. The speedups are shown in Figure 7. The sequential program runs for 53 seconds. The 8-processor speedups using TreadMarks and PVM are 4.46 and 6.17, respectively. Several factors contribute to the 28% difference in performance. The most important reason is the diff requests in TreadMarks. At 8 processors, TreadMarks sends 10 times more messages than PVM. Among the 33742 messages sent in TreadMarks, about 32000 messages are sent due to diff requests and diff transmission. Since the bubblesort threshold is 1024, all of the intermediate subarrays are larger than one page, resulting in multiple diff requests for each subarray, in addition to some false sharing. *Diff accumulation* also occurs in this case because the intermediate subarrays and the work queue migrate among processes.

3.8 Water

Water from the SPLASH [20] benchmark suite is a molecular dynamics simulation. The main data structure in Water is a one-dimensional array of records, in which each record represents a molecule. It contains the molecule's center of mass, and for each of the atoms, the computed forces, the displacements and their first six derivatives. During each time step, both intra- and inter-molecular potentials are computed. To avoid computing all $n^2/2$ pairwise interactions among molecules, a spherical cutoff range is applied.

The parallel algorithm statically divides the array of molecules into equal contiguous chunks, assigning each chunk to a processor. The bulk of the interprocessor communication happens during the force computation phase. Each processor computes and updates the intermolecular force between each of its molecules and each of $n/2$ molecules following it in the array in wrap-around fashion.

In the TreadMarks version, the Water program from the original SPLASH suite is tuned to get better performance. Only the center of mass, the displacements and the forces on the molecules are allocated in shared memory, while the other variables in the molecule record are allocated in private memory. A lock is associated with each processor. In addition, each processor maintains a private copy of the forces. During the force computation phase, changes to the forces are accumulated locally in order to reduce communication. The shared forces are updated after all processors have finished this phase. If a processor i has updated its private copy of the forces of molecules belonging to processor j, it acquires lock j and adds all its contributions to the forces of molecules owned by processor j. In the PVM version, processors exchange displacements before the force computation. No communication occurs until all the pairwise intermolecular forces have been computed, at which time processors communicate their locally accumulated modifications to the forces.

We used two data set sizes, 288 molecules and 1728 molecules, and ran for 5 time steps. The results are shown in Figures 8 and 9. The sequential execution time for the 288-molecule simulation is 12 seconds. The 8-processor speedups for TreadMarks and PVM are 5.04 and 7.23, respectively. With 1728 molecules, the sequential program runs for 435 seconds. TreadMarks and PVM achieve speedups of 7.25 and 7.44 at 8 processors, respectively.

Low computation/communication ratio, separation of synchronization and data communication, and false sharing are the major reasons for the 30% gap in performance at 288 molecules. In PVM, two user-level messages are sent for each pair of processors that interact with each other, one message to read the displacements, and the other message to write the forces. In TreadMarks, extra messages are sent for synchronization and for diff requests to read the displacements or to write the shared forces. After the barrier that terminates the phase in which the shared forces are updated, a processor may fault again when reading the final force values of its own molecules, if it was not the last processor to update those values or if there is false sharing. False sharing causes the processor to bring in updates for molecules that it does not access, and may result in communication with more than one processor if molecules on the same page are updated by two different processors. At 8 processors, false sharing occurs on 7 of the 11.8 pages of the molecule array. Consequently, TreadMarks sends 5028 messages compared to 620 messages in PVM. False sharing also causes the TreadMarks version to send unnecessary data. Another cause of the additional data sent in TreadMarks is *diff accumulation*. Assuming there are n processors, where n is even, the molecules belonging to a processor are modified by $n/2+1$ processors, each protected by a lock. On average, each processor gets $n/2$ diffs. Since all the molecules are not modified by each processor in this case, the diffs are not completely overlapping. Adding both false sharing and diff accumulation, at 8 processors, TreadMarks sends 2.3 times more data than PVM. Increased computation/communication ratio and reduced false sharing cause TreadMarks to perform significantly better at 1728 molecules. TreadMarks sends only 1.3 times more data than PVM, compared to 2.3 times more with 288 molecules.

3.9 Barnes-Hut

Barnes-Hut from the SPLASH [20] benchmark suite is an N-body simulation using the hierarchical Barnes-Hut Method. A tree-structured hierarchical representation of physical space is used. Each leaf of the tree represents a body, and each internal node of the tree represents a "cell", a collection of bodies in close physical proximity. The major data structures are two arrays, one representing the bodies and the other representing the cells. The sequential algorithm loops over the bodies, and for each body traverses the tree to compute the forces acting on it.

In the parallel code, there are four major phases in each time step.

1. MakeTree : Construct the Barnes-Hut tree.
2. Get_my_bodies: Partition the bodies among the processors.
3. Force Computation: Compute the forces on my own bodies.
4. Update: Update the positions and the velocities of my bodies.

Phase 1 is executed sequentially, because running in parallel slows down the execution. In phase 2, dynamic load balance is achieved by using the cost-zone method, in which each processor walks down the Barnes-Hut tree and collects a set of logically consecutive leaves. Most of the computation time is spent in phase 3.

In the TreadMarks version, the array of bodies is shared, and the cells are private. In MakeTree, each processor reads all the shared values in bodies and builds internal nodes of the tree in its private memory. There are barriers after the MakeTree, force computation, and update phases. No synchronization is necessary during the force computation phase. The barrier at the end of the force computation phase ensures that all processors have finished reading the positions of all other processors. In the PVM version, every processor broadcasts its bodies at the end of each iteration, so that each processor obtains all the bodies and creates a complete tree in phase 1. No other communication is required.

We ran Barnes-Hut with 4096 bodies for 6 timesteps. The last 5 iterations are timed in order to exclude any cold start effects. Figure 10 shows the speedups. The sequential program runs for 17 seconds. At 8 processors, PVM and TreadMarks achieve speedups of 3.04 and 2.70 respectively. The low computation to communication ratio and the need for fine-grained communication [19] contribute to the poor speedups on both TreadMarks and PVM. In PVM, the network is saturated at 8 processors, because every processor tries to broadcast at the same time. Diff requests and false sharing are the major reasons for TreadMarks' lower performance. At 8 processors, TreadMarks sends 44% more data and about 222 times more messages than PVM. Although the set of bodies owned by a processor are adjacent in the Barnes-Hut tree, they are not adjacent in memory. Because of false sharing, in MakeTree, each page fault causes the processor to send out diff requests to several processors. For the same reason, during the force computation, a processor may fault on accessing its own bodies, and bring in unwanted data.

3.10 3-D FFT

3-D FFT, from the NAS [2] benchmark suite, numerically solves a partial differential equation using three dimensional forward and inverse FFT's. Assume the input array A is $n_1 \times n_2 \times n_3$, organized in row-major order. The 3-D FFT first performs a n_3-point 1-D FFT on each of the $n_1 \times n_2$ complex vectors. Then it performs a n_2-point 1-D FFT on each of the $n_1 \times n_3$ vectors. Next, the resulting array is transposed into an $n_2 \times n_3 \times n_1$ complex array B and an n_1-point 1-D FFT is applied to each of the $n_2 \times n_3$ complex vectors.

We distribute the computation on the array elements along the first dimension of A, so that for any i, all elements of the complex matrix $A_{i,j,k}$, $0 \leq j < n_2, 0 \leq k < n_3$ are assigned to a single processor. No communication is needed in the first two phases, because each of the n_3-point FFTs or the n_2-point FFTs is computed by a single processor. The processors communicate with each other at the transpose, because each processor accesses a different set of elements afterwards.

In the TreadMarks version, a barrier is called before the transpose. In the PVM version, messages are sent explicitly. To send these messages, we must figure out where each part of the A array goes to, and where each part of the B array needs to come from. These index calculations on a 3-dimensional array are much more error-prone than simply swapping the indices, as in TreadMarks, making the PVM version harder to write.

The results are obtained by running on a $64 \times 64 \times 64$ array of double precision complex numbers for 6 iterations, excluding the time for distributing the initial values at the beginning of program. This matrix size is 1/32 of that specified in the class A problem in the NAS benchmarks. We scaled down the problem because of the limited swap space on the machines available to us. The speedup curves are shown in Figure 11. The sequential execution time is 41 seconds. A speedup of 4.41 is obtained by TreadMarks at 8 processors, which is 81% of the speedup of 5.47 obtained by PVM. Because of release consistency, TreadMarks sends almost the same amount of data as PVM, except for the 6-processor execution. However, because of the page-based invalidate protocol, many more messages are sent in TreadMarks than in PVM. At 8 processors, about 4 megabytes of data are communicated in each transpose. With a page size of 4096 bytes, each transpose therefore requires about 1000 diff requests and responses.

An anomaly occurs at 6 processors, which we attribute to false sharing. Because the matrix size is not a multiple of 6, a page modified by one processor is read by two other processors. Although the two processors read disjoint parts of the page, the same diff is sent to both of them. As a result, in TreadMarks, 14% more messages and 17% more data are sent at 6 processors than at 8 processors.

3.11 ILINK

ILINK [7, 15] is a widely used genetic linkage analysis program that locates specific disease genes on chromosomes. The input to ILINK consists of several family trees. The program traverses the family trees and visits each nuclear family. The main data structure in ILINK is a pool of `genarrays`. A genarray contains the probability of each genotype for an individual. Since the genarray is sparse, an index array of pointers to nonzero values in the genarray is associated with each one of them. A bank of genarrays large enough to accommodate the biggest nuclear family is allocated at the beginning of program, and the same bank is reused for each nuclear family. When the computation moves to a new nuclear family, the pool of genarrays is reinitialized for each person in the current family. The computation either updates a parent's genarray conditioned on the spouse and all children, or updates one child conditioned on both parents and all the other siblings.

We use the parallel algorithm described in Dwarkadas et al. [8]. Updates to each individual's genarray are parallelized. A master processor assigns the nonzero elements in the parent's genarray to all processors in a round robin fashion. After each processor has worked on its share of nonzero values and updated the genarray accordingly, the master processor sums up the contributions of each of the processors.

In the TreadMarks version, the bank of genarrays is shared among the processors, and barriers are used for synchronization. In the PVM version, each processor has a local copy of each genarray, and messages are passed explicitly between the master and the slaves at the beginning and the end of each nuclear family update. Since the genarray is sparse, only the nonzero elements are sent. The diffing mechanism in TreadMarks automatically achieves the same effect. Since only the nonzero elements are modified during each nuclear family update, the diffs transmitted to the master only contain the nonzero elements.

We used the CLP data set [12], with an allele product $2 \times 4 \times 4 \times 4$. The results are shown in Figure 12. The sequential program runs for 1473 seconds. At 8 processors, TreadMarks achieves a speedup of 5.57, which is 93% of the 5.99 obtained by PVM. A high computation-to-communication ratio leads to good speedups and also explains the fact that PVM and TreadMarks are close in performance. However, we were able to identify three reasons for the lower performance of TreadMarks. First, while both versions send only the nonzero elements, PVM performs this transmission in a single message. TreadMarks sends out a diff request and a response for each page in the genarray. For the CLP data set, the size of the genarray is about 16 pages. Second, false sharing occurs in TreadMarks because the nonzero values in the parents' genarrays are assigned to processors in a round robin fashion. In PVM, when the parents' genarrays are distributed, each processor gets only its part of the genarray, but in TreadMarks, a processor gets all the nonzero elements in the page, including those belonging to other processors. The third and final reason for the difference in performance is *diff accumulation*. The bank of genarrays is re-initialized at the beginning of the computation for each nuclear family. Although the processors need only the newly initialized data, TreadMarks also sends diffs created during previous computations.

3.12 Summary

From our experience with PVM and TreadMarks, we conclude that it is easier to program using TreadMarks than using PVM. Although there is little difference in programmability for simple programs, for programs with complicated communication patterns, such as ILINK and 3-D FFT, it takes a lot of effort to figure out what to send and whom to send it to.

Our results show that because of the use of release consistency and the multiple-writer protocol, Tread-

Marks performs comparably with PVM on a variety of problems in the experimental environment examined. These results are corroborated by those in [4], which performed a similar experiment comparing the Munin DSM system against message passing on the V System [6]. For five out of the twelve experiments, TreadMarks performed within 10% of PVM. Of the remaining experiments, Barnes-Hut and to a lesser extent IS-Large exhibit poor performance on both PVM and TreadMarks. With the data sets used, these applications have too low a computation-to-communication ratio for a network of workstations. For the remaining five experiments, the performance differences are between 10% and 30%.

The separation of synchronization and data transfer and the request-response nature of data communication in TreadMarks are responsible for lower performance for all the TreadMarks programs. In PVM, data communication and synchronization are integrated together. The send and receive operations not only exchange data, but also regulate the progress of the processors. In TreadMarks, synchronization is through locks/barriers, which do not communicate data. Moreover, data movement is triggered by expensive page faults, and a diff request is sent out in order to get the modifications. In addition, PVM benefits from the ability to aggregate scattered data in a single message, an access pattern that would result in several miss messages in the invalidate-based TreadMarks protocol.

Although the multiple-writer protocol addresses the problem of simultaneous writes to the same page, false sharing still affects the performance of TreadMarks. While multiple processors may write to disjoint parts of the same page without interfering with each other, if a processor reads the data written by one of the writers after a synchronization point, diff requests are sent to all of the writers, causing extra messages and data to be sent.

In the current implementation of TreadMarks, diff accumulation occurs as a result of several processors modifying the same data, a common pattern with migratory data. Diff accumulation is not a serious problem when the diff sizes are small, because several diffs can be sent in one message.

4 Conclusions

This paper presents two contributions. First, our results show that, on a large variety of programs, the performance of a well optimized DSM system is comparable to that of a message passing system. Especially for problems of practical size, such as ILINK and the Water simulation of 1728 molecules, TreadMarks performs within 10% of PVM. In terms of programmability, our experience indicates that it is easier to program using TreadMarks than using PVM. Although there is little difference in programmability for simple programs, for programs with complicated communication patterns, such as ILINK and 3-D FFT, a lot of effort is required to determine what data to send and whom to send the data to.

Second, we observe four main causes for the lower performance of TreadMarks compared to PVM: the separation of synchronization and data transfer in TreadMarks, additional messages to send diff requests in the invalidate-based TreadMarks protocol, false sharing, and finally diff accumulation for migratory data.

We are currently integrating compiler analysis with the DSM runtime system to alleviate some of these problems. If the compiler can determine future data accesses, prefetching can reduce the cost of page faults and diff requests. Furthermore, in some cases data movement can be piggybacked on the synchronization messages, overcoming the separation of synchronization and data movement. In addition, the compiler can place data in such a way as to minimize false sharing overhead.

Acknowledgments

We would like to thank Rand Hoven from Hewlett-Packard and Stephen Poole from The Performance Group for giving us access to the workstation clusters used in the experiments described. We would also like to thank Nenad Nedelkovic and Edmar Wienskoski who participated in an early version of this project.

References

[1] S. Adve and M. Hill. Weak ordering: A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[2] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report 103863, NASA, July 1993.

[3] B.N. Bershad and M.J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, September 1991.

[4] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related information in distributed shared memory systems. To appear in ACM Transactions on Computer Systems.

[5] 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.

[6] D.R. Cheriton and W. Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 129–140, October 1983.

[7] R. W. Cottingham Jr., R. M. Idury, and A. A. Schäffer. Faster sequential genetic linkage computations. *American Journal of Human Genetics*, 53:252–263, 1993.

[8] 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.

[9] G.A. Geist and V.S. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience*, pages 293–311, June 1992.

[10] 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.

[11] R.J. Harrison. Portable tools and applications for parallel computers. In *International Journal of Quantum Chemistry*, volume 40, pages 847–863, February 1990.

[12] J. T. Hecht, Y. Wang, B. Connor, S. H. Blanton, and S. P. Daiger. Non-syndromic cleft lip and palate: No evidence of linkage to hla or factor 13a. *American Journal of Human Genetics*, 52:1230–1233, 1993.

[13] 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.

[14] 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.

[15] G. M. Lathrop, J. M. Lalouel, C. Julier, and J. Ott. Strategies for multilocus linkage analysis in humans. *Proceedings of National Academy of Science, USA*, 81:3443–3446, June 1984.

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

[17] Message Passing Interface Forum. MPI: A message-passing interface standard, version 1.0, May 1994.

[18] Parasoft Corporation, Pasadena, CA. Express user's guide, version 3.2.5, 1992.

[19] J.P. Singh, J.L. Hennessy, and A. Gupta. Implications of hierarchical n-body methods for multiprocessor architectures. *ACM Transactions on Computer Systems*, 13(2):141–202, May 1995.

[20] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):2–12, March 1992.

Honghui Lu received the B.S. degree in Computer Science and Engineering from Tsinghua University, China, in 1992, and the M.S. degree in Electrical and Computer Engineering from Rice University in 1995. She is currently a computer engineering Ph.D. student under the direction of Professor Willy Zwaenepoel. Her research interests include parallel and distributed systems, parallel computation, and performance evaluation.
e-mail: hhl@cs.rice.edu

Sandhya Dwarkadas received the B.Tech. degree in Electrical and Electronics Engineering from the Indian Institute of Technology, Madras, India, in 1986, and the M.S. and Ph.D. degrees in Electrical and Computer Engineering from Rice University in 1989 and 1993. She is currently a research scientist at Rice University. While at Rice, she has worked on the Rice Parallel Processing Testbed, an execution-driven simulation system, the design of Willow, a high performance parallel architecture, and TreadMarks, a software distributed shared memory system. Her research interests include parallel and distributed systems, parallel computer architecture, parallel computation, simulation methodology, and performance evaluation.
e-mail: sandhya@cs.rice.edu

Alan Cox has worked on both hardware and software for parallel computing over the last nine years. He received the B.S. in Applied Mathematics from Carnegie Mellon University in 1986, and the M.S. and Ph.D. in Computer Science from the University of Rochester in 1988 and 1992, respectively. In 1991, he joined the faculty of Rice University as an Assistant Professor of Computer Science. Some results of his work include a new cache coherence protocol for shared-memory multiprocessors that speeds the execution of parallel programs, innovative systems software for large-scale, shared-memory multiprocessors that simplifies application programming by automating data placement and distribution, and a new algorithm for implementing DSM on workstation networks. This algorithm is used in TreadMarks. In recognition of this work, Prof. Cox was named an NSF Young Investigator in 1994. In addition, he is a co-founder of Parallel Tools, LLC, a Texas-based company founded in 1994 to commercialize the TreadMarks software.
e-mail: alc@cs.rice.edu

Willy Zwaenepoel received the B.S. degree from the University of Gent, Belgium, in 1979, and the M.S. and Ph.D. degrees from Stanford University in 1980 and 1984. Since 1984, he has been on the faculty at Rice University. His research interests are in distributed operating systems and in parallel computation. While at Stanford, he worked on the first version of the V kernel, including work on group communication and remote file access performance. At Rice, he has worked on fault tolerance, protocol performance, optimistic computations, distributed shared memory, and nonvolatile memory.
e-mail: willy@cs.rice.edu