Quantifying the Performance Differences Between PVM and TreadMarks

Honghui Lu

Department of Electrical and Computer Engineering
Rice University
Houston, TX 77005-1892
e-mail: hhl@cs.rice.edu

Sandhya Dwarkadas

Department of Computer Science
University of Rochester
Rochester, NY14627-0226
e-mail: sandhya@cs.rochester.edu

Alan L. Cox, and Willy Zwaenepoel

Department of Computer Science
Rice University
Houston, TX 77005-1892
e-mail: {alc, willy}@cs.rice.edu

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.

“PVM Versus TreadMarks”

Honghui Lu

Computer Science Department
Rice University
Houston, TX 77005-1892
(713) 527-8750 ext 2852

Abstract

This paper compares two systems for parallel programming on networks of workstations: Parallel Virtual Machine (PVM), a message passing system, and TreadMarks, a software distributed shared memory (DSM) system. The eight applications used in this comparison 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) and Traveling Salesman (TSP). Two different input data sets are used for five of the applications. We use two execution environments. The first is a 155Mbps ATM network with eight Sparc-20 model 61 workstations; the second is an eight processor IBM SP/2.

The differences in speedup between TreadMarks and PVM depend mostly on the application, and, only to a much lesser extent, on the platform and the data set used. In particular, the TreadMarks speedup for six of the eight applications is within 15% of that achieved with PVM. For one application, the difference in speedup is between 15% and 30%, and for another, the difference is around 50%.

We identified four important factors that contribute to the lower performance of TreadMarks: 1) extra messages due to the separation of synchronization and data transfer, 2) extra messages to handle access misses caused by the use of an invalidate protocol, 3) false sharing, and 4) *diff accumulation* for migratory data.

We have quantified the effects of the last three factors by measuring the performance gain when each is eliminated. Of the three factors, TreadMarks' use of a separate request message per page of data accessed is the most important. The effect of false sharing is comparatively lower. Reducing diff accumulation benefits migratory data only when the diffs completely overlap. When these performance impediments are removed, all of the TreadMarks programs perform within 25% of PVM, and for six out of eight experiments, TreadMarks is less than 5% slower than PVM.

List of Tables

1 Introduction

Parallel computing on networks of workstations has gained significant attention in recent years. Because workstation clusters use "off the shelf" products, they are cheaper than supercomputers. Furthermore, high-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 must be performed by sending messages over the network. Currently, the prevailing programming model for parallel computing on this platform is message passing, using libraries such as PVM [9], TCGMSG [11] and Express [23]. A message passing standard MPI [22] 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., [28, 4, 2, 19]) 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 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 program's 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 hurt performance.

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 [2], with the most commonly used message passing system, PVM [9]. 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 eight parallel programs to both TreadMarks and PVM: Water and Barnes-Hut from the SPLASH benchmark suite [26]; 3-D FFT, Integer Sort (IS), and Embarrassingly Parallel (EP) from the NAS benchmarks [3]; ILINK, a widely used genetic linkage analysis program [8]; and Successive Over-Relaxation (SOR), and Traveling Salesman Problem (TSP). Two different input sets were used for five of the applications. We ran these programs on eight Sparc-20 model 61 workstations, connected by a 155Mbits per second ATM network, and on an eight processor IBM SP/2.

In terms of programmability we observe the following differences between message passing and shared memory in these applications. The main difficulty with message passing arises from programs with irregular array accesses (ILINK) or extensive use of pointers (Barnes-Hut). Message passing requires a cumbersome and error-prone recoding of these accesses. The same difficulty arises with programs with regular but complicated array accesses (Water and 3-D FFT). Unlike ILINK and Barnes-Hut, however, a compiler might alleviate much of the burden in these programs. For programs with task queues (TSP), the "natural" approach with message passing appears to involve writing two programs, a master and a slave, where the shared memory program is naturally symmetric. The remaining three programs, EP, SOR, and IS, are sufficiently simple that there is not much difference in programmability between message passing and shared memory.

Performance differences between PVM and TreadMarks depend on the application, and to a much smaller extent, on the platform and the data set size, at least for the applications and environments considered in this paper. On both platforms, IS performs significantly worse on TreadMarks than on PVM, showing a speedup of only half that of PVM. On the SPARC/ATM network, the speedups of EP, ILINK, SOR, Water, Barnes-Hut, and 3-D FFT are within 15% of PVM, with TSP lagging by about 30%. On the IBM SP/2, 3-D FFT and TSP trade places, with TSP now performing within 15% and 3-D FFT lagging 30%. The relative differences for the other applications remain the same as on the SPARC/ATM platform.

Communication costs have been identified as the primary source of overhead in software DSM implementations. In an earlier study of the performance of TreadMarks [14], execution times were broken down into various components. Memory management and consistency overhead were shown to account for 3% or less of execution time for all applications. In contrast, the percentage of time spent in communication-related operations, either execution time for sending and receiving messages or idle time waiting for some remote operation to complete, accounted for 5 to 55% of the overall execution time, depending on the application.

In explaining the performance differences between PVM and TreadMarks, we therefore focus on differences in the amount of communication between the two systems. More messages and more data are sent in TreadMarks, as a result of 1) extra messages due to the separation of synchronization and data transfer, 2) extra messages to handle access misses caused by the use of an invalidate protocol, 3) false sharing, and 4) *diff accumulation* for migratory data.

This paper extends the results presented by Lu et al. [20], and quantifies the effect of the last three factors by measuring the performance gain when each factor is eliminated. Because the separation of synchronization and data transfer is a fundamental characteristic of the shared memory model, there is no way to assess its contribution to performance without completely deviating from the shared memory model.

The results show that the largest contribution to the difference in performance between TreadMarks and PVM comes from PVM's ability to use a single message to move a large amount of data, while TreadMarks pages in data one page at a time. By modifying TreadMarks to transfer more than one page at a time the number of messages is reduced substantially, with an attendant improvement in performance. The elimination of false sharing, by careful layout and access of data structures, also reduces message count and data size, but not to the same extent as allowing TreadMarks to move amounts of data larger than a page. Finally, diff squashing addresses the diff accumulation problem by combining overlapping diffs in one, reducing message size. It only helps in Integer Sort, where the communication/computation ratio is high, and the diffs overlap completely.

After these modifications to the TreadMarks programs, all of them perform within 25% of PVM, and for six out of nine experiments, TreadMarks is less than 5% slower than PVM.

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 explains our methodology to quantify the contribution of each factor causing extra communication in TreadMarks. Section 4 gives an overview of the experimental results. Section 5 discusses the performance of the different applications. Section 6 discusses related work. Section 7 concludes the paper.

2 PVM Versus TreadMarks

2.1 PVM

PVM [9] provides primitives to send and receive messages, and pack and unpack data. The user data must be *packed* before being dispatched. The *pack* either copies user data into a send buffer, or keeps pointers to user data. An *unpacked* copies received message from the receive buffer to 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 must match the corresponding pack calls in type and number of items.

PVM provides the user only 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, and returns a pointer to the receive buffer. The nonblocking receive returns immediately. If the expected message is present, it returns a pointer to the receive buffer, as with the blocking receive. Otherwise, the nonblocking receive returns a null pointer. A 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, a blocking receive can be called for the same message.

2.2 TreadMarks

TreadMarks [2] is a use-level software DSM system that runs on most Unix systems. We used TreadMarks version 1.0.1 in our experiments.

2.2.1 TreadMarks Interface

TreadMarks provides primitives similar to those used in hardware shared memory machines: 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* version of *release consistency* (RC) [10] and a multiple-writer protocol 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.

The *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.

The execution of each process is divided into *intervals*. A new interval begins every time a process 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 *happen-before-1* [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. TreadMarks then sends diff requests to the subset of processors whose most recent interval is not preceded by the most recent interval of another processor.

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. 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.

2.3 Differences in Performance Between PVM and TreadMarks

There are several reasons why TreadMarks is slower than PVM. 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 and barriers, which do not communicate data.

PVM also benefits from the ability to aggregate scattered data in a single message, an access pattern that would result in several miss messages in TreadMarks' invalidate protocol. Each access miss in TreadMarks is triggered by a page fault, and a diff request and response are sent in order to propagate the modifications.

False sharing still affects the performance of TreadMarks. Although multiple processors may write to disjoint parts of the same page without interfering with each other, when 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 for *migratory* data. Migratory data is shared sequentially by a set of processors [4, 27]. Each processor has exclusive read and write access for a time. Accesses to migratory data are protected by locks in TreadMarks. Each time a processor accesses migratory data, it must see all the preceding modifications. In TreadMarks, this is implemented by fetching all diffs created by processors who have modified the data since the current processor's last access. In case the diffs overlap, this implementation causes more data to be sent than just fetching the most recent diff. Although all the overlapping diffs can be obtained from one processor, diff accumulation still results in more messages when the sum of the diff sizes exceeds the maximum size of a UDP message. Since the maximum UDP message size is 64Kbytes, extra messages due to diff accumulation are not a serious problem.

In addition to differing amounts of communication, TreadMarks also incurs the cost for detecting and recording modifications to shared memory. This cost includes the overhead of memory protection operations, page faults as a result of memory protection violations, twinning and diffing, and the maintenance of timestamps and write notices. Earlier work [14] has demonstrated that in current networking environments this cost is relatively small compared to the communication overhead. We therefore concentrate on the differences in communication, and refer the reader to our earlier paper [14] for a detailed account of consistency overhead.

3 Methodology

We tried to quantify how much each of the aforementioned factors contributed to TreadMarks' performance. Three of them are assessed – lack of bulk transfer, false sharing, and diff accumulation. Because the separation of synchronization and data transfer is a fundamental characteristic of the shared memory model, there is no way to assess its effect on performance without completely deviating from the shared memory model. The contribution of each factor is measured by the performance gain when the factor is eliminated. When several factors contribute significantly to an application's performance, we also measured the aggregate effect of eliminating all of them simultaneously.

The effect of bulk transfer is achieved by defining the TreadMarks page size to be a multiple of the hardware page size. By increasing the TreadMarks page size, on each page fault, a larger block of shared memory is updated, avoiding separate diff requests for each hardware page in this block. For each application, we use the page size which results in the best result. In general, a larger page size may increase the degree of false sharing. Fortunately, for the applications used in this study that benefit from bulk data transfer, the page size could be increased without introducing additional false sharing.

To reduce false sharing, we modified the shared data layout and the data access pattern of the applications in a way that does not significantly alter program behavior. For applications with static data partitioning, we padded each processor's data to page boundaries, eliminating all false sharing. For applications such as TSP, Barnes-Hut and ILINK, which have dynamic access patterns, it is impossible to completely eliminate false sharing without changing the program's behavior. In these cases, we relied on knowledge of the program's access patterns to modify the data layout in such a way to substantially reduce false sharing.

Diff squashing addresses the *diff accumulation* problem. Except where false sharing occurs, diffs are created in a lazy fashion. A diff is not created for a modified page until some processor requests that diff to update its copy of the page. If this request also asks for older diffs, our diff squashing procedure compares each of the older diffs to the new diff, and the parts covered by the new diff are truncated.

We are not proposing that programmers hand-tune their TreadMarks programs using these methods. We are using them here solely to indicate the contributions of various sources of communication overhead. We believe, however, that some of the overheads that were identified can be addressed automatically using new run-time techniques or via compiler support (see Sections 6 and 7).

4 Overview of Experimental Results

4.1 Experimental Testbed

We use two experimental platforms for measurements. The first platform is an 8-node cluster of Sparc-20 model 61 workstations, each with 32 megabytes of main memory, connected by a 155Mbps ATM switch. On this platform, TreadMarks 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.

Our second experimental environment is an 8-processor IBM SP/2 running AIX version 3.2.5. Each processor is a thin node with 64 Kbytes of data cache and 128 Mbytes of main memory. Interprocessor communication is accomplished over IBM's high-performance two-level cross-bar switch. On this platform, TreadMarks is implemented on top of the MPL reliable message passing layer, and we use PVMe, a version of PVM optimized for the IBM SP/2 and also implemented on top of MPL.

We chose these platforms for the following reason. The SPARC/ATM platform is typical of the current generation of "networks of workstations" that use traditional network interfaces. Access to the network interface is through the operating system. The SP/2 is meant to represent the next generation, in which the application may directly access the network interface, thereby significantly reducing the communication overhead. Some basic characteristics of both platforms are given in Table 1.

4.2 Applications

We ported eight parallel programs to both TreadMarks and PVM: Water and Barnes-Hut from the SPLASH benchmark suite [26]; 3-D FFT, IS, and EP from the NAS benchmarks [3]; ILINK, a widely used genetic linkage analysis program [8]; and SOR, and TSP.

The execution times for the sequential programs, without any calls to PVM or TreadMarks, are shown in Table 2. This table also shows the problem sizes used for each application. On the IBM SP/2 we were able to run some applications with larger data sizes. Main memory limitations prevented us from running larger data sets on the SPARC/ATM network.

4.3 Speedups

Table 3 shows the 8-processor speedups of PVM and TreadMarks on both platforms. The speedup is computed relative to the sequential program execution times on each platform given in Table 2. Table 3 also shows the relative performance of TreadMarks compared to PVM. Table 4 shows total memory usage in both systems for all of the applications and data sizes.

As can be seen in Table 3, performance differences between PVM and TreadMarks depend on the application, and to a much smaller extent, on the platform and the data set size, at least for the applications and environments considered in this paper. On both platforms, IS performs significantly worse on TreadMarks than on PVM, showing a speedup of only half of that of PVM. On the SPARC/ATM network, the speedups of EP, ILINK, SOR, Water, Barnes-Hut, and 3-D FFT are with 15% of PVM, with TSP lagging by about 30%. On the

		SPARC/ATM	IBM SP/2
		(msec.)	
TreadMarks	8-processor barrier	2.85	0.82
	2-processor lock	1.14	0.35
	3-processor lock	1.47	0.52
	Empty diff page fault	1.47	1.28
	Full page diff page fault	2.84	2.11
	Memory protection	0.04	(ave.) 0.14
	Signal delivery	0.06	0.39
PVM/PVMe	Empty message round trip	1.31	0.38
	Max bandwidth without copying	8.6 MB/sec.	29.4 MB/sec.
	Max bandwidth with copying	7.2 MB/sec.	21.0 MB/sec.

Table 1 Characteristics of the Experimental Platforms

Program	Problem Size	Time (sec.)	
		SPARC/ATM	IBM SP/2
EP	2^{28}	2021	1733
ILINK	CLP	1364	653
SOR	1024×4096, 50 iterations	125	39
	2048×6144, 100 iterations		118
Water-1728	1728 molecules, 5 iterations	622	413
	4096 molecules, 5 iterations		2344
TSP	19 cities	109	109
	24 cities		170
Barnes-Hut	16384 bodies	122	107
	32768 bodies		242
3-D FFT	$64 \times 64 \times 64$, 6 iterations	51	15
	$128 \times 128 \times 64$, 6 iterations		62
IS	$N = 2^{21}$, $B_{max} = 2^{15}$, 9 iterations	12	7

Table 2 Data Set Sizes and Sequential Execution Time of Applications

Program		SPARC/ATM			IBM SP/2		
		PVM	TreadMarks		PVM	TreadMarks	
EP		7.99	7.99	(100%)	8.00	8.00	(100%)
ILINK		5.83	5.01	(86%)	5.71	5.23	(92%)
SOR	(1024x4096)	7.53	7.28	(97%)	7.56	6.24	(83%)
	(2048x6144)				7.64	6.31	(83%)
Water	(1728)	7.59	7.35	(97%)	8.07	7.65	(95%)
	(4096)				8.07	7.98	(99%)
TSP	(19)	7.94	5.51	(69%)	7.48	7.21	(96%)
	(24)				6.88	7.27	(106%)
Barnes-Hut	(16384)	4.64	4.01	(86%)	6.19	5.87	(95%)
	(32768)				6.40	5.95	(93%)
3-D FFT	(64x64x64)	5.12	4.72	(92%)	5.20	3.73	(72%)
	(128x128x64)				4.88	3.54	(72%)
IS	(21-15)	4.00	1.70	(42%)	4.58	2.42	(53%)

Table 3 8-Processor Speedups and Relative Performance under PVM and TreadMarks

Program		PVM	TreadMarks
EP		18.8	25.6
ILINK		25.9	35.1
SOR	(1024x4096)	35.2	58.4
	(2048x6144)	68.1	92.1
Water	(1728)	19.9	29.5
	(4096)	21.5	32.9
TSP	(19)	19.6	26.8
	(24)	20.1	25.8
Barnes-Hut	(16384)	21.3	35.6
	(32768)	23.8	41.6
3-D FFT	(64x64x64)	35.1	49.1
	(128x128x64)	98.2	142.7
IS	(21-15)	16.5	53.7

Table 4 8-Processor Memory Usage under PVM and TreadMarks (megabytes)

IBM SP/2, 3-D FFT and TSP trade places, with TSP now performing within 15% and 3-D FFT lagging 30%. The relative differences for the other applications remain the same as on the SPARC/ATM platform. For all but IS, memory requirements for TreadMarks exceed those of PVM by 25% to 40%. For IS, the difference is 70%, because of the high amount of twin and diff space required.

The similarity between the results on the two platforms results from a combination of some of the characteristics on each platform. On one hand, lower latency and higher bandwidth on the IBM SP/2 switch causes the extra communication in TreadMarks to have a relatively smaller effect on performance. On the other hand, the longer interrupt latency and the higher cost of memory management operations puts TreadMarks at a disadvantage on this architecture. Only for TSP and 3-D FFT does the PVM vs. TreadMarks tradeoff change noticeably between platforms. For TSP, this change appears to be largely accidental, resulting from the non-deterministic nature of the search algorithm in the program. For 3-D FFT, the superior floating point performance on the IBM SP/2 results in a much lower sequential execution time (see Table 2). As a result, the extra communication in TreadMarks has, relatively speaking, a larger effect on the IBM SP/2, causing a lower speedup.

4.4 Factors Contributing to TreadMarks Performance

For the SPARC/ATM platform, we quantify the effect of removing the various performance impediments from TreadMarks. Table 5 presents speedups, and Tables 6 and 7 provide figures for the number of messages and the amount of data exchanged, which will be used in explaining the speedup numbers (see Section 5). 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 messages, and the total amount of data communicated. Figures for the IBM SP/2 platform are qualitatively the same and are not included.

With the exception of IS, most of the differences in speedup and in communication requirements between TreadMarks and PVM are a result of PVM's ability to aggregate large

Program	PVM	TreadMarks	TreadMarks Bulk Transfer	TreadMarks No FS	TreadMarks Diff Squash
EP	7.99	7.99 (100%)	– –	– –	– –
ILINK	5.83	5.01 (86%)	5.59 (96%)	– –	5.00 (86%)
SOR	7.53	7.28 (97%)	7.53 (100%)	– –	– –
Water-1728	7.59	7.35 (97%)	– –	7.37 (97%)	7.36 (97%)
TSP	7.94	5.51 (69%)	– –	6.08 (76%)	5.50 (69%)
Barnes-Hut	4.64	4.01 (86%)	5.28 (114%)	4.56 (98%)	– –
3-D FFT	5.12	4.72 (92%)	4.99 (98%)	– –	– –
IS	4.00	1.70 (42%)	– –	– –	2.88 (72%)

Table 5 8-Processor Speedups for PVM, TreadMarks, and Various Modifications of TreadMarks on the SPARC/ATM platform

Program	PVM	TreadMarks	TreadMarks Bulk Transfer	TreadMarks No FS	TreadMarks Diff Squash
EP	7	69	–	–	–
ILINK	6615	255001	85063	–	255074
SOR	1400	7034	4220	–	–
Water-1728	620	9033	–	8275	9011
TSP	1384	18169	–	12793	18122
Barnes-Hut	280	144038	41104	41444	–
3-D FFT	1610	13349	3180	–	–
IS	1008	9996	–	–	9975

Table 6 8-Processor Message Totals for PVM, TreadMarks, and Various Modifications of TreadMarks on the SPARC/ATM platform

Program	PVM	TreadMarks	TreadMarks Bulk Transfer	TreadMarks No FS	TreadMarks Diff Squash
EP	0.3	43	–	–	–
ILINK	47583	119933	116808	–	105136
SOR	11474	619	462	–	–
Water-1728	9123	21194	–	19298	11199
TSP	37	3025	–	3390	2685
Barnes-Hut	50551	54968	53721	39170	–
3-D FFT	25690	25973	25956	–	–
IS	16515	51842	–	–	17374

Table 7 8-Processor Data Totals for PVM, TreadMarks, and Various Modifications of TreadMarks on the SPARC/ATM platform (Kilobytes)

amounts of data in a single message. Doing the equivalent thereof in TreadMarks leads to substantial performance improvements for four applications (ILINK, SOR, Barnes-Hut, and 3-D FFT). For IS, diff accumulation is the main performance impediment in TreadMarks, as can be seen from the improvements resulting from diff squashing. For the other applications in which diff accumulation occurs, the high computational overhead of diff squashing causes performance to be adversely affected. Finally, avoiding false sharing has only a limited effect.

5 Discussion of Individual Applications

In this section we discuss the implementation of the applications in terms of PVM and TreadMarks. We identify the applications for which there is a substantial difference in programmability, and we point out the reasons for the difference. In terms of performance, we again focus on the performance of the applications on the SPARC/ATM platform, as the results for the IBM SP/2 are qualitatively the same.

5.1 EP

The Embarrassingly Parallel program comes from the NAS benchmark suite [3]. 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 sequential program runs for 2021 seconds. Both TreadMarks and PVM achieve a speedup of 7.99 using 8 processors, because compared to the overall execution time, the communication overhead is negligible.

5.2 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 1024×4096 matrix of floating point numbers for 51 iterations. With this problem size each shared red or black row occupies two 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 our test the edge elements are initialized to 1, and all the other elements to 0.

The sequential program runs for 125 seconds. At 8 processors, the TreadMarks version and the PVM version achieve speedups of 7.28 and 7.53, respectively. There is only a 3% difference between TreadMarks and PVM, because of the low communication rate in SOR, and the use of lazy release consistency in TreadMarks. The number of messages is 4 times higher in TreadMarks than in PVM. For n processors, PVM sends $2 \times (n-1)$ messages at the end of each phase. In each red or black phase, 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). As a result of diffing in TreadMarks, much less data is sent by TreadMarks than by PVM because most of the pages remain zero.

SOR 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.

To measure the effect of multiple diff requests, we increase TreadMarks page size to 8192 bytes, so that only one diff request and reply are sent in paging in the red or black elements in a row. This reduces the number of messages sent in TreadMarks by 40%, from 7034 to 4220, and TreadMarks only sends 2 times more messages than PVM. Consequently, the performance gap between TreadMarks and PVM shrinks from 4% to zero, and both of them have a speedup of 7.53.

5.3 Integer Sort

Integer Sort (IS) [3] 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 own keys, and writes the result in a private array of buckets. Next, the processors compute the global array of buckets by adding the corresponding elements in each private array of buckets. Finally, all processors rank their keys according to the global array of buckets. To obtain good parallelism, the bucket array is divided equally into n blocks, where n is the number of processes. The global buckets are computed in n steps. In each step, a processor works on one of the blocks, and moves on to another one in the next step.

The global array of buckets is shared in the TreadMarks version. There are n locks, protecting modifications to each of the n blocks of the global bucket array. In step i of the n steps calculating the sum, processor pid acquires lock $(pid + i) \bmod n$ and works on the corresponding block. A barrier synchronizes all processors after the updates. Each processor then reads the final result in the shared array of buckets and ranks its keys. In the PVM version, processors compute the global bucket array in the same order as in TreadMarks. At the end of each step i, a processor sends the result to the next processor in line. After the final step, the last processor modifying the block broadcasts the result to all others.

We sorted 2^{21} keys ranging from 0 to 2^{15} for 9 iterations. We did not try the 2^{23} keys specified in the NAS benchmarks, because it does not fit into a single machine's memory. The sequential execution time for IS is 12 seconds. The 8 processor speedups for PVM and TreadMarks are 4.0 and 1.7, respectively.

TreadMarks sends 9975 messages, about 9 times more than PVM. The extra messages are mostly due to multiple diff requests. There are 2^{15} integers in the global bucket array. At eight processors, each of the eight blocks contains 4 pages. Therefore, each time a processor adds to a block of the global bucket, TreadMarks sends 4 diff requests and responses, while PVM handles the transmission of the block with a single message exchange.

The extra data in TreadMarks comes from *diff accumulation*, because blocks in the global bucket array migrate among the processors. Moreover, the diffs by different processors completely overlap with each other in this case. On each page fault taking place when a processor modifies the global array of buckets, the processor receives diffs created by all the preceding processors. If there are any succeeding processors, the processor will fault on the same page again when it reads the final result in the global bucket. Each page fault brings in the diffs made by all the succeeding processors. Assuming the array 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 $n \times (n-1) \times b$.

Diff accumulation is the most important factor. Without diff accumulation, the data sent in TreadMarks is reduced by 2/3, from 50 megabytes to 16 megabytes, only 7% more than PVM. As a result, TreadMarks' speedup increases from 1.70 to 2.88, which is 72% of PVM.

Since the performance of IS is bounded by the communication bandwidth, the contribution of multiple diff requests cannot be measured with the presence of diff accumulation. By using the 16-kilobyte page size in addition to diff squashing, the number of diff requests and replies is reduced by 3/4, and message total is reduced to 5943, 60% of the original TreadMarks. The effect is that the 8-processor speedup increases to 3.38, 85% of PVM. (This result does not appear in Table 3 because it can not be measured separately.)

5.4 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 lower bound of the most promising path from the priority queue is longer than current best tour. 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` and update to the shortest path are protected by two distinct locks. The PVM version uses a master-slave arrangement. One processor runs both the master and the slave process, while the remaining processors run only a slave process. The master keeps all the major data structures in its private memory, executes `get_tour`, and keeps track of the shortest tour. 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 sequential program runs for 109 seconds. The 8-processor speedup of TreadMarks is 5.51, 69% of the speedup of 7.94 obtained by PVM. At 8 processors, TreadMarks sends 12 times more messages and 80 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, which only take 2 messages. In TreadMarks, however, all the major data structures migrate among the processors. In `get_tour`, it takes at least 3 page faults to obtain the tour pool, the priority queue, and the tour stack.

False sharing affects TreadMarks because a 4096-byte page can hold up to 27 tours. When a processor pops off a tour from the stack, it can bring in data for other tours on the same page even though it may not actually access them.

Because of diff accumulation, a processor can get up to $(n-1)$ diffs on each page fault, where n is the number of processors in the system. Due to the random access pattern on the tour pool and the priority queue, the diffs are not completely overlapping,

Furthermore, there is some contention for the lock protecting `get_tour`. On average, at 8 processors, each process spends 2 out of 20 seconds waiting at lock acquires.

We eliminate false sharing on the tour pools by keeping a separate tour pool and tour stack for each processor. The result shows that 30% of the messages are attributed to false sharing. In the absence of false sharing, TreadMarks performance improves from 5.51 to 6.08, which is 76% of PVM.

Diff accumulation accounts for 11% of the data sent in TreadMarks, or 340 kilobytes, but it contributes little to TreadMarks performance. With the high speed networks we use, message size is a secondary factor in deciding communication cost compared with number of messages.

5.5 Water

Water from the SPLASH [26] 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. Inter-molecular forces are computed by interacting each molecule with each of the $n/2$ molecules following it in the array in wrap-around fashion, and applying a spherical cutoff range.

The parallel algorithm statically divides the array of molecules into equal contiguous blocks, assigning each block to a processor. The bulk of the interprocessor communication takes place during the inter-molecular force computation.

In the TreadMarks version, the Water program from the original SPLASH suite is tuned to get better performance. The molecule record is divided into two parts, in which the center of mass, the displacements and the forces on the molecules are allocated in shared memory, while the other variables are allocated in private memory. During force computation, updates to the shared forces are buffered in a private force array. The shared forces are updated at the end, protected by a lock associated with each processor. If a processor's molecules interact with the molecules belonging to processor j, it acquires lock j and adds all its local contributions to the molecules owned by processor j. In the PVM version, processors exchange displacements before the force computation. No communication occurs until all the pairwise inter-molecular forces have been computed, at which time processors communicate their locally accumulated modifications to the forces.

We chose the parameters to be 1728 molecules and 5 time steps. The sequential program runs for 622 seconds. For this problem size, this application has a high computation to communication ratio. At 8 processors, despite the fact that TreadMarks sends 14 times more messages and 1.3 times more data than PVM, TreadMarks and PVM achieve speedups of 7.35 and 7.59, respectively.

The performance difference is mainly caused by synchronization, and the invalidate protocol used by TreadMarks. 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 the synchronizations. Besides, a processor has to request for the page when it reads the displacements, writes to the shared forces, and reads the final force values of its own molecules.

An important source of the additional data sent in TreadMarks is *diff accumulation*. Assuming there are n processors, where n is even, each page containing the shared forces migrates among $n/2 + 1$ processors. On average, each processor gets $n/2$ diffs. Because of the cutoff range, the diffs are not completely overlapping.

False sharing exists only in the boundary pages of each processor's block. Because the force data owned by each processor contains 3.8 pages, force sharing is rare.

To eliminate false sharing, each processor allocates its own part of the shared force and displacement arrays in shared memory. Each allocation is padded to an integral of page size. False sharing only constitutes 8% of the messages sent in TreadMarks, and has little effect on TreadMarks' performance.

Although diff accumulation is responsible for 47% of the total data sent under TreadMarks, the performance is hardly affected because of the high computation to communication ratio.

5.6 Barnes-Hut

Barnes-Hut from the SPLASH [26] 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 16384 bodies for 6 timesteps. The last 5 iterations are timed in order to exclude any cold start effects. The sequential program runs for 122 seconds. At 8 processors, PVM and TreadMarks achieve speedups of 4.64 and 4.01 respectively. The low computation to communication ratio and the need for fine-grained communication [25] contribute to the poor speedups on both TreadMarks and PVM. TreadMarks sends 513 times more messages than PVM at 8 processors. This is the result of both false sharing and multiple diff requests. Although the set of bodies owned by a processor are adjacent in the Barnes-Hut tree, they are not adjacent in memory. This results in false sharing that causes each page fault to send diff requests to all 8 processors. Moreover, since the body array spans 368 pages, it takes a processor 368 page faults and corresponding requests and replies to obtain it.

We increase the TreadMarks page size to 16Kbytes, the maximum currently allowed, to reduce the number of diff requests. Because every page is shared by all 8 processors, using a larger page size does not worsen the false sharing. The message count drops by 71% as a result, and TreadMarks speedup is improved from 4.01 to 5.28, 14% faster than the speedup of 4.64 obtained by PVM. This anomaly occurs because TreadMarks uses UDP, while PVM uses TCP, which has higher overhead.

We reduce false sharing by reorganizing the bodies in memory. After all the processors have found their own bodies in the first iteration, the bodies are copied so that all the bodies belonging to the same processor are adjacent in body array. Because the position of bodies changes very slowly, the set of bodies owned by a processor remains almost the same over the next 5 iterations. With reduced false sharing, TreadMarks sends 71% less messages and 29% less data at 8 processors. TreadMarks speedup is increased to 4.56, only 2% slower than PVM.

5.7 3-D FFT

3-D FFT, from the NAS [3] 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 in order to enable the program to execute on one machine without paging. The sequential execution time is 51 seconds. A speedup of 4.72 is obtained by TreadMarks at 8 processors, which is 92% of the speedup of 5.12 obtained by PVM. Because of release consistency and the absence of false sharing, TreadMarks sends almost the same amount of data as PVM. However, because of the page-based invalidate protocol, 7.3 times more messages are sent in TreadMarks than in PVM.

To reduce multiple diff requests, we increase TreadMarks page size to 8192 bytes, which is the largest page size that does not incur false sharing. The number of messages sent in TreadMarks drops to twice that in PVM. Consequently, TreadMarks' speedup increases from 4.72 to 4.99, less than 2% lower than the speedup of 5.12 obtained in PVM.

5.8 ILINK

ILINK [6, 18] is a widely used genetic linkage analysis program that locates specific disease genes on chromosomes. The input to ILINK consists of several family trees. The program traverses the family trees and visits each nuclear family. The main data structure in ILINK is a pool of `genarrays`. A genarray contains the probability of each genotype for an individual. Since the genarray is sparse, an index array of pointers to non-zero values in the genarray is associated with each one of them. A bank of genarrays large enough to accommodate the biggest nuclear family is allocated at the beginning of execution, and the same bank is reused for each nuclear family. When the computation moves to a new nuclear family, the pool of genarrays is reinitialized for each person in the current family. The computation either updates a parent's genarray conditioned on the spouse and all children, or updates one child conditioned on both parents and all the other siblings.

We use the parallel algorithm described by Dwarkadas et al. [8]. Updates to each individual's genarray are parallelized. A master processor assigns the non-zero elements in the parent's genarray to all processors in a round robin fashion. After each processor has worked on its share of non-zero 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 the bank of genarrays, and messages are passed explicitly between the master and the slaves at the beginning and the end of each nuclear family update. The user program explicitly packs the non-zero elements in the genarrays into a message, and unpacks the message accordingly. The diffing mechanism in TreadMarks automatically achieves the same effect. Since only the non-zero elements are modified during each nuclear family update, the diffs transmitted to the master only contain the non-zero elements.

We used the CLP data set [12], with an allele product $2 \times 4 \times 4 \times 4$. 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 non-zero 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 non-zero 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 non-zero 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.

The TreadMarks page size is increased to 16 kilobytes to reduce diff requests. This reduces 67% of the messages, and increased TreadMarks speedup from 5.01 to 5.59, which is 96% of PVM. Because of the high computation to communication ratio and the fact that only 12% of the data sent in TreadMarks are attributed to diff accumulation, diff accumulation has little affect on TreadMarks. It is hard to measure the effect of false sharing because of the dynamic access pattern, and because a processor accesses completely different data each time.

5.9 Summary

5.9.1 Programmability

From our experience with PVM and TreadMarks, we conclude that it is easier to achieve correctness and efficiency using TreadMarks. Although there is little difference in programmability for simple programs, such as EP, SOR and IS, for programs with complicated communication patterns, such as Water, 3-D FFT, Barnes-Hut, ILINK and TSP, it takes more effort to write a correct and efficient message passing program.

In the TreadMarks version of Water, a single call to a lock and a barrier synchronize the updates to the shared force array. Another call to the barrier after updating the displacements makes sure that all processors will receive the new displacement values in the next iteration. In the PVM version, however, instead of inserting a synchronization call, the programmer needs to compute the source and destination of each piece of data, copy the data to and from the message buffer, and issue send and receive calls. While the TreadMarks code has 1842 lines, another 440 lines are required in the PVM code.

For 3-D FFT, the array transpose in TreadMarks consists of simple operations to switch indices, and a call to the barrier before the transpose. In the PVM version, one must envision how data is moved in a 3-D transpose, and generate communication calls accordingly. The PVM version has 160 more lines than the TreadMarks version. Moreover, the index calculations on a 3-D array are more error-prone than simply swapping the indices, as in

TreadMarks.

In the PVM version of Barnes-Hut, we let each processor broadcast all its nodes. This simple algorithm works fine with a small number of processors, but would have serious problems when scaled to a larger cluster. However, writing the message passing code that exactly selects which nodes are going to be accessed by what processor would be quite involved. In TreadMarks, instead, a barrier call causes processors to page in those nodes they access.

In ILINK, by adding a barrier between different phases of the computation, TreadMarks automatically transmits the non-zero elements in the `genarray`. In the PVM version, if we take the simple approach of sending all values, including the zeroes, the resulting performance becomes worse than that obtained by TreadMarks. Exactly picking out the non-zero values adds significantly to the complexity of the PVM code. The TreadMarks version has 10902 lines. The PVM version has an additional 432 lines, including the code to pick out the non-zero genarray elements.

For TSP, because TreadMarks provides a shared memory interface, we can write a simple 811-line program in which all the processes are equivalent. In the PVM version, two different programs are written for the master and the slave processes, which increases the code size by half, to 1253 lines.

5.9.2 Performance

Our results show that because of the use of release consistency and the multiple-writer protocol, TreadMarks performs comparably with PVM on a variety of problems in the experimental environment examined. For six out of the eight experiments, TreadMarks performs within 15% of PVM. For the remaining four experiments, TreadMarks lags behind PVM for 31% and 68% for TSP and IS, respectively.

After eliminating three of the prime factors that slow down TreadMarks, all of the TreadMarks programs perform within 25% of PVM, and for six out of eight experiments, Tread-

Marks is less than 5% slower than PVM. Of the three factors we experimented with, the effect of multiple diff requests is the most significant. Four out of eight experiments benefits from bulk transfer, with two of them gaining over 10%. This can be attributed to the large data size and coarse granularity of these applications. Three of the applications perform better with the elimination of false sharing, but all of the improvements are less than 10%. Diff squashing only reduces the data totals, which is a second order effect in TreadMarks overheads. As a result, it is profitable only for IS, where diffs completely overlap.

6 Related Work

Our study distinguishes itself from most related work by being, with the exception of Carter et al. [4], the first study to compare message passing to *software* distributed shared memory, implemented on top of message passing. We are thus evaluating the cost of layering shared memory in software on top of message passing, in contrast to the studies that evaluate message passing and shared memory as two architectural models implemented in hardware. In contrast to the work on Munin [4], we use lazy rather than eager release consistency. It has been demonstrated that lazy release consistency leads to lower communication requirements and better performance [14]. Furthermore, our study is done on common Unix platforms and using a well-known message passing system.

Among the architectural studies comparing message passing and shared memory, we cite two recent articles, namely Chandra et al. [5] and Klaiber and Levy [15]. Both of these are simulation studies, while our results are derived from measurements of an implementation. Chandra et al. [5] compares four applications, running either with a user-space message passing or with a full-map invalidate shared memory coherence protocol. All other simulation parameters, such processor and network characteristics and number of processors, are kept the same. For three of their applications, shared memory has the same performance as message passing. For these applications, the software overhead of the message passing layers compensates for the extra communication in the shared memory programs. For their fourth application, extra communication causes shared memory to perform about 50% worse than message passing.

Klaiber and Levy [15] compare the communication requirements of data-parallel programs on message passing and shared memory machines. We focus instead on execution times, and use the communication requirements as a means to explain the differences in execution times. Also, their data-parallel programs are compiled by two different compilers, one for message passing and one for shared memory. The results may therefore be influenced by differences in the quality of the code generated by the two compilers.

Having recognized the advantages and drawbacks of shared memory and message passing, several groups have recently proposed machine designs that integrate both architectural models [16, 17, 24]. Various compiler techniques can also be used to remedy some of the deficiencies of shared memory recognized in this study. For instance, Eggers and Jeremiassen [13] discuss compiler transformations to reduce the effect of false sharing, and Dwarkadas et al. [7] evaluate compiler support for communication aggregation, merging data and synchronization, and reduction of coherence overhead.

Finally, there have a variety of papers comparing implementations of individual applications in shared memory and message passing, including, e.g., hierarchical N-body simulation [25] and VLSI cell routing [21].

7 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 non-trivial size, such as ILINK and Water, TreadMarks performs within 15% 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 Water, 3-D FFT, Barnes-Hut, ILINK and TSP, a lot of effort is required to determine what data to send and whom to send the data to. Especially for programs with complicated or irregular array accesses or with data structures accessed through pointers, the message passing paradigm is harder to use and more error-prone.

Second, we observe four main causes for the lower performance of TreadMarks compared to PVM, namely 1) extra messages due to the separation of synchronization and data transfer, 2) extra messages to handle access misses caused by the invalidate protocol, 3) false sharing, and 4) diff accumulation for migratory data.

Without deviating from a shared memory model, we designed some experiments to measure the performance contribution of each of the last three factors. The results show that the effect of the extra messages to handle access misses is the most significant. Four out of eight of the applications benefit from the elimination of this factor, with two of them gaining over 10%. This can be attributed to the large data size and coarse granularity of these applications. The elimination of false sharing improves TreadMarks performance for three of the experiments, but all of the improvements are less than 10%. Eliminating the diff accumulation only reduces the data totals, which is a second order effect in TreadMarks overheads. As a result, it is profitable only for IS, where diffs completely overlap. Without the three factors that slow down TreadMarks, all of the TreadMarks programs perform within 25% of PVM, and for six out of eight experiments, TreadMarks is less than 15% slower than PVM.

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] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations.

IEEE Computer, 29(2):18–28, February 1996.

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

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

[5] S. Chandra, J.R. Larus, and A. Rogers. Where is time spent in message-passing and shared-memory programs? In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 61–73, October 1994.

[6] 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.

[7] S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[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] T.E. Jeremiassen and S. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the 5th Symposium on the Principles and Practice of Parallel Programming*, July 1995.

[14] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. An evaluation of software-based release consistent protocols. *Journal of Parallel and Distributed Computing*, 29:126–141, October 1995.

[15] A.C Klaiber and H.M. Levy. A comparison of message passing and shared memory architectures for data parallel languages. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 94–106, April 1994.

[16] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B. Lim. Integrating message-passing and shared-memory: Early experience. In *Proceedings of the 1993 Conference on the Principles and Practice of Parallel Programming*, pages 54–63, May 1993.

[17] J. Kuskin and D. Ofelt et al. The Stanford FLASH multiprocessor. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[18] 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.

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

[20] H. Lu, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. Message passing versus distributed shared memory on networks of workstations. In *Proceedings SuperComputing '95*, December 1995.

[21] M. Martonosi and A. Gupta. Tradeoffs in message passing and shared memory implementations of a standard cell router. In *1989 International Conference on Parallel Processing*, pages 88–96, August 1989.

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

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

[24] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[25] 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.

[26] 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.

[27] W.-D. Weber and A. Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.

[28] M.J. Zekauskas, W.A. Sawdon, and B.N. Bershad. Software write detection for distributed shared memory. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, pages 87–100, November 1994.

Biographies

Honghui Lu received the B.S. degree from Tsinghua University, China, in 1992, and the M.S. degree 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, parallelizing compilers, and parallel computation.

e-mail: hhl@cs.rice.edu *URL:* http://www.cs.rice.edu/˜hhl

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. From 1992-1996, she was a research scientist at Rice University. She is currently on the faculty at the University of Rochester. Her research interests include parallel and distributed systems, parallel computation and computer architecture, compiler-run-time integration, simulation methodology, and performance evaluation.

e-mail: sandhya@cs.rochester.edu *URL:* http://www.cs.rochester.edu/u/sandhya

Alan Cox received the B.S. degree from Carnegie Mellon University in 1986 and the M.S. and Ph.D. degrees from the University of Rochester in 1988 and 1992, respectively. He is currently on the faculty at Rice University. His research interests include parallel and distributed systems, distributed garbage collection, and multi-computer architectures.

e-mail: alc@cs.rice.edu *URL:* http://www.cs.rice.edu/˜alc

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 *URL:* http://www.cs.rice.edu/˜willy