Journal of Parallel and Distributed Computing **60**, 1512–1530 (2000)
doi:10.1006/jpdc.2000.1658, available online at http://www.idealibrary.com on IDEAL®

OpenMP for Networks of SMPs

Y. Charlie Hu

Department of Computer Science, Rice University, Houston, Texas 77005
E-mail: ychu@cs.rice.edu

Honghui Lu

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

and

Alan L. Cox and Willy Zwaenepoel

Department of Computer Science, Rice University, Houston, Texas 77005
E-mail: alc@cs.rice.edu, willy@cs.rice.edu

Received September 23, 1999; revised April 11, 2000; accepted April 26, 2000

In this paper, we present the first system that implements OpenMP on a network of shared-memory multiprocessors. This system enables the programmer to rely on a single, standard, shared-memory API for parallelization within a multiprocessor and between multiprocessors. It is implemented via a translator that converts OpenMP directives to appropriate calls to a modified version of the TreadMarks software distributed shared-memory (SDSM) system. In contrast to previous SDSM systems for SMPs, the modified TreadMarks system uses POSIX threads for parallelism within an SMP node. This approach greatly simplifies the changes required to the SDSM in order to exploit the intranode hardware shared memory.

We present performance results for seven applications (Barnes-Hut, CLU, and Water from SPLASH-2, 3D-FFT from NAS, Red-Black SOR, TSP, and MGS) running on an SP2 with four four-processor SMP nodes. A comparison between the thread implementation and the original implementation of TreadMarks shows that using the hardware shared memory within an SMP node significantly reduces the amount of data and the number of messages transmitted between nodes and consequently achieves speedups that are up to 30% better than the original versions. We also compare SDSM against message passing. Overall, the speedups of multithreaded TreadMarks programs are within 7–30% of the MPI versions.

Key Words: OpenMP shared memory programming; POSIX threads; networks of SMPs; software distributed shared memory.

0743-7315/00 $35.00

1. INTRODUCTION

The OpenMP Application Programming Interface (API) is an emerging standard for parallel programming on shared-memory multiprocessors. It defines a set of program directives and a library for run-time support that augment standard C, C++, and Fortran [19, 20]. In contrast to MPI [12], a message-passing API, OpenMP facilitates an incremental approach to the parallelization of sequential programs: The programmer can add a parallelization directive to one loop or subroutine of the program at a time. Unlike POSIX threads [4], OpenMP specifically addresses the needs of scientific programming, such as support for Fortran and data parallelism. An earlier proposal for a standard, shared-memory API for scientific programming, ANSI X3H5, was never formally adopted, leading vendors to create their own similar but subtly different programming models. OpenMP consolidates these different models into a single syntax and semantics and finally delivers the long-awaited promise of single-source portability for shared-memory parallelism.

This paper reports on the first system that implements OpenMP on a *network* of shared-memory multiprocessors. This system enables the programmer to rely on a single, standard, shared-memory API for parallelization within a multiprocessor *and* between multiprocessors. Previously, the only standard APIs available on this type of platform were message-passing standards. In our system, OpenMP's program directives are processed by a source-to-source translator that is constructed from the SUIF Toolkit [1]. In effect, the translator converts each OpenMP directive into the appropriate calls to a modified version of the TreadMarks software distributed shared-memory (SDSM) system [2]. The translated source is a standard C or Fortran 77 program that is compiled and linked with the modified TreadMarks system.

In its simplest form, running TreadMarks on a network of SMPs could be achieved by simply executing a (Unix) process on each processor of each multiprocessor node and having all of these processes communicate through message passing. This approach requires no changes to TreadMarks, and we will therefore refer to it as the *original version*. This version, however, fails to take advantage of the hardware shared memory on the multiprocessor nodes. In order to overcome this limitation, we have built a new version of TreadMarks, in which we use POSIX threads to implement parallelism within a multiprocessor. As a result, the OpenMP threads within a multiprocessor share a single address space. We will refer to this system as the *thread version*. Our approach is distinct from previous SDSM systems for networks of SMPs, such as Cashmere-2L [18] and HLRC-SMP [13], which use (Unix) processes to implement parallelism within an SMP. Each of these processes has a separate address space, although the shared-memory regions (and some other data structures) are mapped shared between the processes. We will refer to such a system as a *process version*.

The use of a single address space within a multiprocessor has pluses and minuses. On the positive side, it reduces the number of changes to TreadMarks to support multithreading on a multiprocessor. For example, the data within an address space on a multiprocessor is shared by default. Furthermore, a page protection operation

by one thread applies to the other threads within the same multiprocessor; the operating system maintains the coherence of the page mappings automatically.

On the negative side, using a single address space within a multiprocessor makes it more difficult to provide uniform sharing of memory between threads on the same node and threads on different nodes. Under POSIX threads, an application's global variables are shared between threads within a multiprocessor, but under TreadMarks they are private with respect to threads on a different multiprocessor.[1] However, since we provide the OpenMP API to the programmer, these differences are hidden by our OpenMP translator (see Section 4).

We measure our OpenMP system's performance on an IBM SP2 with multiprocessor nodes. The machine has four nodes, and each node has four processors. We use seven applications: Barnes-Hut, CLU, and Water from SPLASH-2 [21], 3D-FFT from NAS [3], Red-Black SOR, TSP, and MGS. We compare the results for our OpenMP system to two alternatives: OpenMP with the original TreadMarks and MPI. In both of these cases, message passing is used even between processes on a single node.

Our results show that using hardware shared memory within an SMP node significantly reduces the amount of data and the number of messages transmitted. Consequently, the speedups improve up to 30% over the original TreadMarks implementation. In addition, we found that the multithreaded TreadMarks performs 1.5–5 times fewer page protection operations. Our experiments also show that the multithreaded TreadMarks programs incur 1.2–5 times fewer page faults than their single-threaded counterparts.

We also compare SDSM against message passing. Overall, the speedups of multithreaded TreadMarks programs are within 7–30% of the MPI versions.

The remainder of this paper is organized as follows. Section 2 presents an overview of the OpenMP API. Section 3 presents an overview of the original TreadMarks system and describes the modifications made to support OpenMP on a network of shared-memory multiprocessors. Section 4 describes the source-to-source translator for OpenMP. Section 5 evaluates our system's overall performance and compares it to the original TreadMarks and to MPI. Section 6 discusses related work. Section 7 summarizes our conclusions.

2. THE OPENMP API

The OpenMP API [19, 20] defines a set of program directives that enable the user to annotate a sequential program to indicate how it should be executed in parallel. There are three kinds of directives: parallelism/work sharing, data environment, and synchronization. In C and C++, the directives are implemented as `#pragma` statements, and in Fortran 77 and 90 they axe implemented as comments. We only explain the directives relevant to this paper and refer interested readers to the OpenMP standard [19, 20] for the full specification.

[1] This issue is not specific to TreadMarks. The use of a single address space on a node would give rise to similar nonuniformities in sharing whithin and across nodes with other SDSM systems.

OpenMP is based on a fork-join model of parallel execution. The sequential code sections are executed by a single thread, called the *master thread.* The parallel code sections are executed by all threads, including the master thread.

The fundamental directive for expressing parallelism is the `parallel` directive. It defines a *parallel region* of the program that is executed by multiple threads. All of the threads perform the same computation, unless a *work sharing* directive is specified within the parallel region. Work sharing directives, such as `for`, divide the computation among the threads. For example, the `for` directive specifies that the iterations of the associated loop should be divided among the threads so that each iteration is performed by a single thread. The `for` directive can take a `schedule` clause that specifies the details of the assignment of the iterations to threads. Schedules can specify assignments such as round robin or block. OpenMP also defines shorthand forms for specifying a parallel region containing a single work sharing directive. For example, the `parallel for` directive is shorthand for a `parallel` region that contains a single `for` directive.

The data environment directives control the sharing of program variables that are defined outside the scope of a parallel region.[2] The data environment directives include: `shared`, `private`, `firstprivate`, `reduction`, and `threadprivate`. Each directive is followed by a list of variables. Variables default to `shared`, which means shared among all the threads in a parallel region. A `private` variable has a separate copy per thread. Its value is undefined when entering or exiting a parallel region. A `firstprivate` variable has the same attributes as a `private` variable except that the private copies are initialized to the variable's value at the time the parallel region is entered. The `reduction` directive identifies reduction variables. According to the standard, reduction variables must be scalar, but we have extended the standard to include arrays. Finally, OpenMP provides the `threadprivate` directive for named common blocks in Fortran and global variables in C and C++. `Threadprivate` variables are private to each thread, but they are global in the sense that they are defined for all parallel regions in the program, unlike `private` variables which are defined only for a particular parallel region.

The synchronization directives include `barrier`, `critical`, and `flush`. A `barrier` directive causes a thread to wait until all of the other threads in the parallel region have reached the barrier. After the `barrier`, all threads are guaranteed to see all modifications made before the barrier. A `critical` directive restricts access to the enclosed code to only one thread at a time. When a thread enters a critical section, it is guaranteed to see all modifications made by all of the threads that entered the critical section earlier. The `flush` directive specifies a point in the program at which all threads are guaranteed to have a consistent view of the variables named in the `flush` directive or of the entire memory if no variables are specified.

3. TREADMARKS

TreadMarks [2] is a user-level SDSM system that runs on most Unix and Windows NT-based systems. It provides a global shared address space on top of

[2] The variables defined inside of a parallel region are implicitly private.

physically distributed memories. Under TreadMarks, parallel threads synchronize via primitives similar to those used in hardware shared-memory machines: barriers and locks. In C, the program has to call the `Tmk_malloc` routine to allocate shared variables in the shared heap. In Fortran, shared variables are placed in a common block loaded in a standard location.

3.1. Implementation Overview

Memory coherence and synchronization are the key functions performed by TreadMarks.

3.1.1. *Memory coherence.* TreadMarks relies on user-level memory-management support provided by the operating system to detect accesses to shared memory at the granularity of a page. A *lazy invalidate* version of *release consistency* (RC) [8] and a multiple-writer protocol are employed to reduce the amount of communication involved in implementing the shared-memory abstraction.

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 thread p to become visible to another thread q only when a subsequent release by p becomes visible to q via some chain of synchronization events. In practice, this model allows a thread to buffer multiple writes to shared data in its local memory until a synchronization point is reached.

With the multiple-writer protocol, two or more threads can simultaneously modify their own copies 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 run-length 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: The releasing thread informs the acquiring thread about which pages have been modified, causing the acquiring thread to *invalidate* its local copy of each modified page. A thread incurs a page fault on the first access to an invalidated page and obtains the modifications necessary to update its copy from the previous releasers.

3.1.2. *Synchronization.* Barrier arrivals are implemented as releases, and barrier departures are implemented as acquires. Barriers have a centralized manager. At a barrier arrival, each thread sends a release message to the manager and waits for a departure (acquire) message. The manager broadcasts a barrier departure message to all of the threads after the last thread has arrived at the barrier.

The two primitives for mutex locks are lock release and lock acquire. Each lock has a statically assigned manager. The manager records which thread has most recently requested the lock. All lock acquire requests are sent to the manager and, if necessary, forwarded by the manager to the thread that last requested the lock.

3.2. Modifications for OpenMP

To support OpenMP-like programming models, recent versions of TreadMarks include `Tmk_fork` and `Tmk_join` primitives, specifically tailored to the fork-join style of parallelism expected by OpenMP and most other compilers [1] targeting shared memory. To minimize overhead, all threads are created at the start of a program's execution. During sequential execution, the slave threads are blocked waiting for the next `Tmk_fork` issued by the master.

3.3. Modifications for Networks of Multiprocessors

The modified version of TreadMarks uses POSIX threads to implement parallelism within a multiprocessor. Hence, the OpenMP threads within a multiprocessor share a *single* address space. This has many advantages and a few disadvantages in both the implementation of TreadMarks and its interface.

3.3.1. *Implementation issues.* By using POSIX threads, data are shared by default among the processors within a single node, and coherence is maintained automatically by the hardware. Thus, we did not have to modify TreadMarks to enable the sharing of application data or its own internal data structures between processors within the same node. We did, however, have to modify TreadMarks to place some data structures, such as message buffers, in thread-private memory.

Synchronizing access by the processors within a node to TreadMarks' internal data structures was straightforward. The critical sections within TreadMarks were already guarded by synchronization because incoming data and synchronization requests occur asynchronously, interrupting the application. Thus, with one exception, we simply changed the existing synchronization to work with POSIX threads. The exception is that we added a per-page mutex to allow greater concurrency in the page fault handler.

The synchronization functions provided by TreadMarks to the program were modified to combine the use of POSIX threads-based synchronization between processors within a node and the existing TreadMarks implementation between nodes. Thus, the program can continue to use a single API, that of TreadMarks, for synchronization.

Our last change to the implementation was in the memory coherence mechanism. Similar to Millipede [9], we added a second mapping at a different address within each node's address space for the TreadMarks supported shared data heap/common block, i.e., the memory that is shared between nodes. The first, or original, mapping is used exclusively by the application. The TreadMarks code uses the second mapping, which permits read and write access at all times, to update shared-memory pages so that the application's mapping can remain invalid while the update is in progress. This ensures that another thread cannot read or modify the page until the update is complete.

In fact, the use of two mappings reduces the number of `mprotect`, or page protection, operations performed by TreadMarks, even on a single-processor node. For example, in the original TreadMarks, a read access to an invalid page would result in two `mprotect` operations: one to enable write access in order to update

the page and another to make the page read-only after the update. In the modified version, only the latter `mprotect` operation is performed. The second mapping eliminates the need for the first `mprotect` operation.

Finally, because of our use of a single address space, the operating system automatically maintains the coherence of the page mappings in use by the different processors within a node. Furthermore, an `mprotect` by one thread within a node applies to the other threads. In contrast, systems such as Cashmere-2L [181, which use Unix processes instead of POSIX threads, must perform the same `mprotect` in each process's address space. The reason is that `mprotect` only applies to the calling process's address space, even if the underlying memory is shared between address spaces.

3.3.2. *Interface Issues.* Our use of POSIX threads had one undesirable effect on the TreadMarks interface. Under POSIX threads, global variables are shared, whereas in the original TreadMarks API, global variables are private. Thus, in our modified version of TreadMarks, global variables are shared between threads within a multiprocessor but are private with respect to threads on a different multiprocessor. Rather than attempting to solve this problem in the run-time system, we chose to address it in the OpenMP translator where a solution is straightforward.

4. THE OPENMP TRANSLATOR

The OpenMP to TreadMarks translation process is relatively simple, because TreadMarks already provides a shared memory API across the network. First, the OpenMP synchronization directives translate directly to TreadMarks synchronization operations. Second, the compiler translates the code sections marked with `parallel` directives to fork-join code. Third, it implements the data environment directives in ways that work with both TreadMarks and POSIX threads, hiding the interface issues discussed in Section 3.3.2 from the programmer.

4.1. Implementing Parallel Directives

To translate a sequential program annotated with parallel directives into a fork-join parallel program, the translator encapsulates each parallel region into a separate subroutine. This subroutine also includes code, generated by the compiler, that allows each thread to determine, based on its thread identifier, which portions of a parallel region it needs to execute. At the beginning of a parallel region, the master thread passes a pointer to this subroutine to the slave threads. Pointers to `shared` variables and initial values of `firstprivate` variables are copied into a structure and passed to the slaves at the same time.

4.2. Implementing Data Environment Directives

Variables accessed within a parallel region default to `shared`. If a global variable is annotated `threadprivate`, it cannot be annotated again within a parallel region. Thus, the translator allocates all global variables on the shared heap unless they are annotated `threadprivate`.

For each `threadprivate` global variable, the compiler allocates an array of n copies of the global variable, where n is the number of threads per node. Each reference to the global variable is replaced by a reference to the array, specifically, a reference to the element corresponding to the thread's (local) id.

In TreadMarks, a thread's stack is kept in private memory. Thus, variables declared within a procedure that are accessed within a parallel region must be moved to the shared heap. In addition, variables declared within a procedure and passed by reference to another procedure are moved to the shared heap because the translator cannot prove that such a variable will not be used in a parallel region. Storage for these variables is allocated at the beginning of the procedure and freed at the end.

Implementing `private` variables is straightforward: whenever a variable is annotated private within a parallel region, its definition is duplicated in the procedure generated by the compiler that encapsulates the parallel region. Because each thread calls this procedure after the fork, these variables will be allocated on the private stack of each thread.

5. PERFORMANCE

5.1. Platform

Our experimental platform is an IBM SP2 consisting of four SMP nodes. Each node contains four IBM PowerPC 604 processors and 1 Gbyte of memory. All of the nodes are running AIX 4.2.

5.2. Applications and Their OpenMP Implementations

We use seven applications in this study: SPLASH-2 Barnes-Hut, NAS 3D-FFT, SPLASH-2 CLU and Water, Red-Black SOR, TSP, and MGS. Table I summarizes the problem size, the sequential running time, and the parallelization directives used in the OpenMP implementation of each application. The sequential running times are used as the basis for the speedups reported in the next section.

TABLE I

Application, Problem Size, Sequential Execution Time, and Parallelization Directive(s) in the OpenMP Programs

Application	Size, iterations	Sequential time (s)	OpenMP parallel directives
Barnes	65536	158.0	parallel regin
3D-FFT	$128 \times 128 \times 64$, 10	65.2	parallel for
CLU	2048×2048, block: 32	86.9	parallel region
Water	4096, 4	760.3	parallel for/region
SOR	$8K \times 4K$, 20	149.0	parallel for
TSP	19 cities, $-$r14	248.1	parallel region
MGS	$2K \times 2K$	563.3	parallel for

Barnes: Barnes-Hut from SPLASH-2 is an *N*-body simulation code using the hierarchical Barnes-Hut method. A shared tree structure is used to represent the recursively decomposed subdomains (cells) of the three-dimensional physical domain containing all of the particles. The other shared data structure is an array of particles corresponding to the leaves of the tree. Each iteration is divided into two steps.

1. Tree building: A single thread reads the particles and rebuilds the tree.
2. Force evaluation: All treads participate., First, they divide the particles by traversing the tree in the Morton ordering (a linear ordering of the points in higher dimension) of the cells. Specifically, the *i*th thread locates the *i*th segment. The size of a segment is weighted according to the workload recorded from the previous iteration. Then, each of the threads performs the force evaluation for its particles. This involves a partial traversal of the tree. Overall, each thread reads a large portion of the tree.

In OpenMP, the force evaluation is parallelized using the `parallel` region directive.

3D-FFT: 3D-FFT from the NAS benchmark suite solves a partial differential equation using three-dimensional forward and inverse FFT. The program has three shared arrays of data elements and an array of checksums. The computation is decomposed so that every iteration includes local computation and a global transpose.

In OpenMP, the data parallelism in the local computation and the global transpose is expressed using the `parallel for` directive.

CLU: Continuous LU from SPLASH-2 performs LU factorization without pivoting. The matrix is divided into square blocks that are distributed among processors in a round-robin fashion. Furthermore, the blocks owned by the same processor are allocated consecutively in shared memory to minimize false sharing.

In OpenMP, the parallelism along both dimensions of the matrix is expressed using the `parallel` region directive.

Water: Water from SPLASH-2 is a molecular dynamics simulation. The main data structure in Water is a one-dimensional array of molecules. During each time step, both intra- and intermolecular potentials are computed. The parallel algorithm statically divides the array of molecules into equally sized contiguous blocks, assigning each block to a thread. The bulk of the interprocessor communication results from synchronization that takes place during the intermolecular force computation.

In OpenMP, the evaluation of intramolecule potentials requires no interactions between molecules and is parallelized using the `parallel for` directive. The evaluation of intermolecule potentials is parallelized using the `parallel` region directive. Each thread is assigned a subset of the molecules. It accumulates the results of the force computation on subsets of molecules assigned to other threads into its private memory during the computation. Afterward, all the threads synchronize with each other and sum up the contributions to their own subsets of molecules in a staggered fashion.

SOR: Red-Black Successive Over-Relaxation is a method for solving partial differential equations by iterating over a two-dimensional array. In every iteration, each of the array elements is updated to the average of the element's four nearest neighbors.

These data parallel operations are expressed in OpenMP using the `parallel for` directive.

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. A thread repeatedly dequeues the most promising path from the priority queue and either extends it by one city and enqueues the new path or takes the dequeued path and tries all permutations of the remaining cities.

In OpenMP, the threads are created using the `parallel` region directive. Accesses to the priority queue are synchronized using the `critical` directive.

MCS: Modified Gramm–Schmidt (MGS) computes an orthonormal basis for a set of N-dimensional vectors. During the ith iteration, the algorithm first normalizes the ith vector sequentially and then, in parallel, makes all vectors $j > i$ orthogonal to vector i. Vectors are assigned to threads in a cyclic manner to balance the load. All threads synchronize at the end of each iteration.

In OpenMP, the normalization of each vector is performed by the master thread, and the parallel updates are expressed using the `parallel for` directive with a cyclic schedule.

5.3. *Results*

We first compare the performance of the OpenMP programs translated into TreadMarks programs modified to use POSIX threads within an SMP node (OpenMP/thread) against the performance of those same programs translated into original TreadMarks programs using processes (OpenMP/original). In the latter, processes on the same node communicate via message passing instead of using the hardware shared memory.

We then compare the OpenMP/thread and OpenMP/original versions of the applications against MPI versions of the same applications. We use the MPICH (http://www.mcs.anl.gov /mpi/mpich) implementation of MPI.

Figure 1 shows the speedups for the OpenMP/original programs with four processes per node, OpenMP/thread programs with four threads per node, and MPI programs with four processes per node on the four-node SP2. Figures 2 and 3 compare the amount of data and the number of messages transmitted in the above three versions of the applications. For each application, the values on the y-axis are normalized with respect to the OpenMP/thread version and are truncated at 4. The absolute numbers are listed in Table II.

For the MPI version, we report both the total number of messages and the number of messages that actually cross node boundaries.

5.3.1. *OpenMP/original versus OpenMP/thread.* In terms of relative speedups, the applications can be categorized into four groups. The first group, consisting of

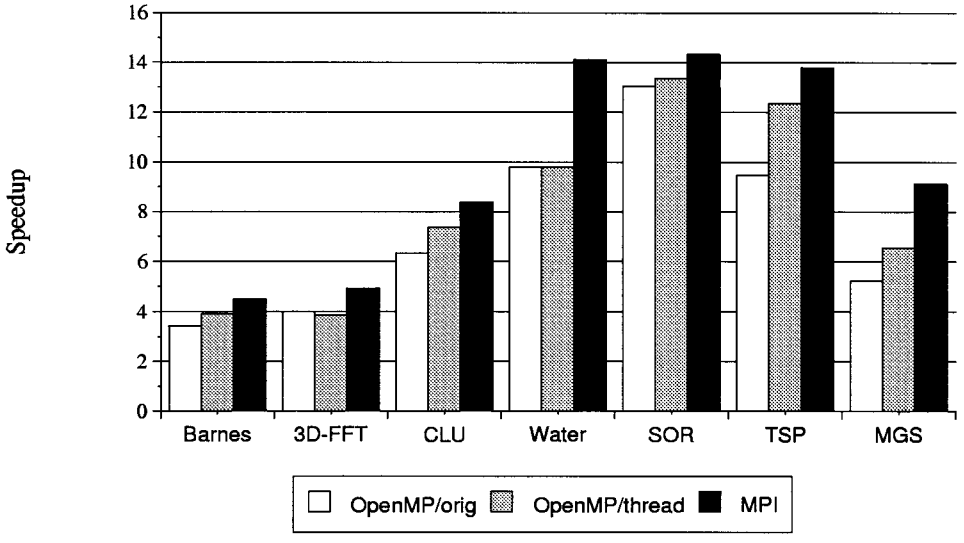

FIG. 1. Speedup comparison between the OpenMP-original, OpenMP/thread, and MPI versions of the applications on an SP2 with four-processor SMP nodes.

Barnes, CLU, and MGS, has low to moderate computation to communication ratios. For these programs, the 2- to 5-fold reduction in the amount of data transmitted results in significant speedups. The second group, consisting of Water and SOR, has high computation to communication ratios. In this case, a 4.5- to 9-fold reduction in data loads to little improvement in running time. TSP forms the third group. It also has a high computation to communication ratio. The OpenMP/thread version does, however, achieve performance improvement because an update of the current shortest path is immediately seen by all threads on the same node which effectively reduces the number of paths to be expanded. FFT

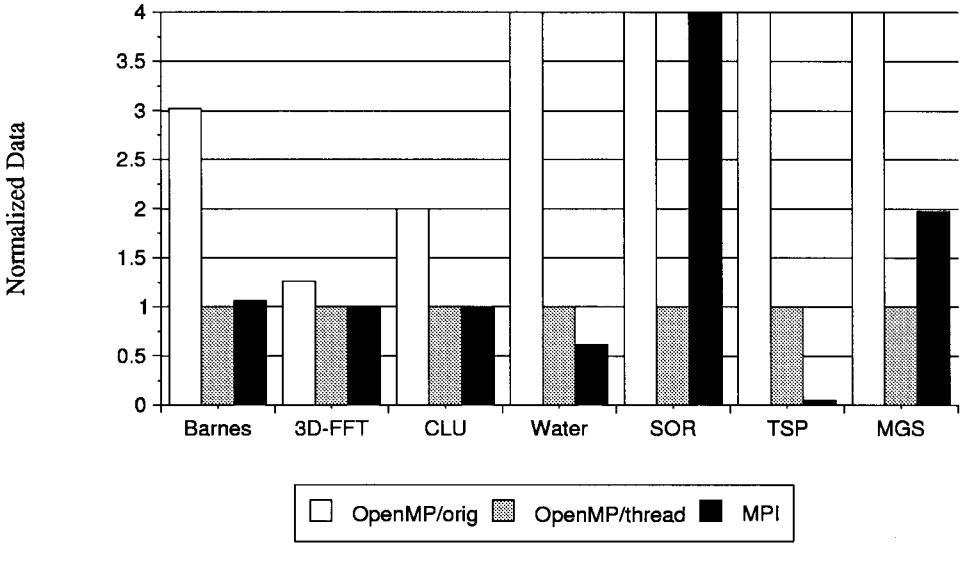

FIG. 2. Normalized data comparison between the OpenMP/original, OpenMP/thread, and MPI versions of the applications on an SP2 with four-processor SMP nodes. The *y*-axis is truncated at 4.

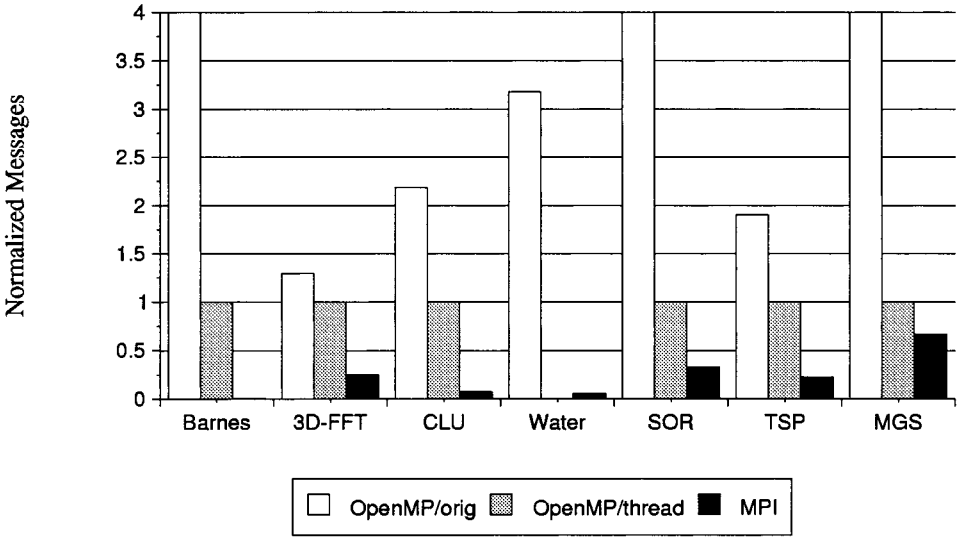

FIG. 3. Normalized messages comparison between the OpenMP/original, OpenMP/thread, and MPI versions of the applications on an SP2 with four-processor SMP nodes. The *y*-axis is truncated at 4.

TABLE II

Amount of Data and Number of Messages Transmitted in the OpenMP/original, OpenMP/thread, and MPI Versions of the Applications on an SP2 with Four Four-Processor SMP Nodes

Application	OpenMP/ original	OpenMP/ thread	MPI	
			Total	Off-node
		Data (Mbytes)		
Barnes	593.3	196.5	259.7	207.8
3D-FFT	159.4	126.5	157.3	125.8
CLU	102.2	51.2	102.2	51.1
Water	192.3	42.7	34.6	26.0
SOR	0.64	0.07	9.8	2.0
TSP	2.8	0.55	0.03	0.026
MGS	508.6	102.2	251.6	201.3
		Messages		
Barnes	1478908	156371	720	576
3D-FFT	40975	31694	9750	7800
CLU	28895	13239	1860	930
Water	78402	24667	1776	1344
SOR	3637	735	1200	240
TSP	9227	4853	1256	1070
MGS	184583	37041	30720	24576

forms the fourth group, where we see a slight slowdown for the OpenMP/thread code. The slowdown happens in the transpose stage where all processors request data from one processor at a time, invoking large numbers of request handlers on that processor's node. The reason for the slowdown is due to AIX's inefficient implementation of page protection operations as explained below.

Overall, compared to OpenMP/original, the OpenMP/thread programs send less data, from a low of 26% less data for 3D-FFT to a high of 9.1 times less data for SOR, and fewer messages, from a low of 29% fewer messages for 3D-FFT to a high of 9.5 times fewer messages for Barnes.

Figure 4 compares the number of times that the `mprotect` operation is performed in the original and the thread versions of the translated OpenMP programs on four SMP nodes. The values on the *y*-axis are normalized with respect to the thread versions. The detailed numbers are listed in Table III. First, the OpenMP/thread programs with one thread per node perform 25–56% fewer `mprotect` operations than the corresponding OpenMP/original versions with one process per node, indicating that the alias mapping (see Section 3.3.1) reduces the number of `mprotect` operations independent of any multithreading effects. Second, the OpenMP/thread programs with four threads per node perform 1.9–6.2 times fewer `mprotect` operations than the OpenMP/original codes with four processes per node. To separate the contributions to this reduction from using double mapping and from using multithreading, we further measure the number of `mprotect` operations in running the OpenMP/thread programs with 16 processes, each containing one thread, on the four nodes. The comparison with the above two 16-way parallelism versions are summarized in Table IV. The comparison between Thrd/16 × 1 results and the Orig/16 × 1 results shows that the use of double

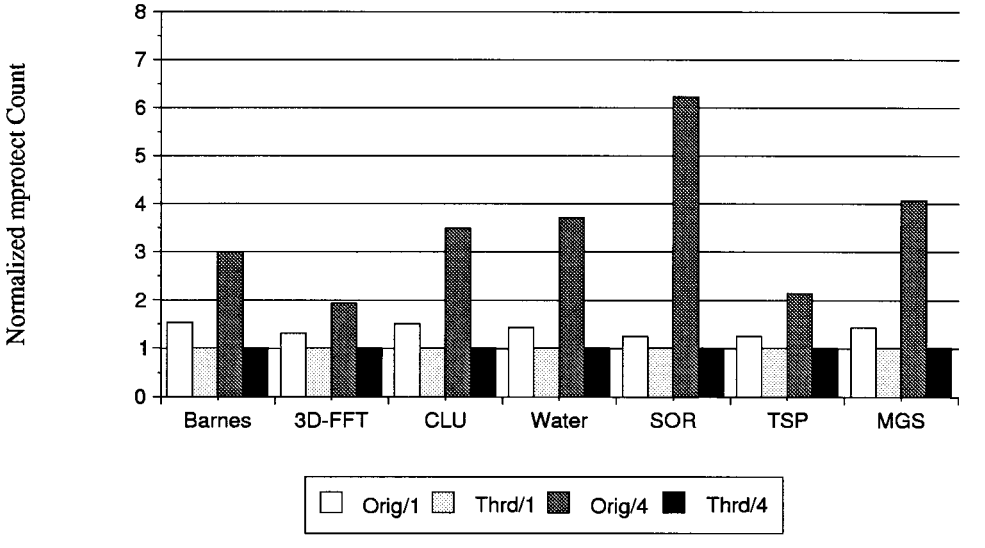

FIG. 4. Normalized `mprotect` count comparison between the OpenMP/original and OpenMP/thread versions of the applications on four four-processor SMP nodes. Orig/1 and Orig/4 denote OpenMP/original with one and four processes on a node, and Thrd/1 and Thrd/4 denote OpenMP/thread with one and four threads on a node, respectively. Orig/1 and Orig/4 are normalized with respect to Thrd/1 and Thrd/4, respectively.

TABLE III

Number of `mprotect` Operations, Average `mprotect` Time, Number of Page Faults, and DSiffs in the OpenMP/original and OpenMP/thread Versions of the Applications Running on Four-Processor SMP Nodes

Application	Orig/1	Thrd/1	Orig/4	Thrd/4
		`mprotect` count		
Barnes	112452	73322	320743	107114
3D-FFT	65650	50200	97690	50588
CLU	12411	8253	29015	8308
Water	27244	19071	102879	27691
SOR	1209	969	6037	969
TSP	6947	5529	11628	5438
MGS	30730	21511	86154	21118
		Average `mprotect` time (μs)		
Barnes	49.6	63.9	65.6	115.5
3D-FFT	82.0	137.6	62.5	644.9
CLU	82.6	258.8	61.8	555.5
Water	57.8	83.8	72.2	128.4
SOR	216.9	339.8	142.5	708.5
TSP	37.6	51.5	42.7	85.2
MGS	66.3	183.7	67.5	234.8
		Page fault count		
Barnes	55810	55810	161565	83349
3D-FFT	30860	30860	39020	31155
CLU	4158	4158	12460	6230
Water	13533	13523	46130	28705
SOR	480	480	2400	480
TSP	2895	2889	4794	4047
MGS	14336	14336	40346	32404
		diff count		
Barnes	29941	29941	87651	44333
3D-FFT	15404	15404	19370	15501
CLU	4095	4095	4095	2079
Water	5090	5090	13017	7890
SOR	240	240	1200	240
TSP	1394	1394	1599	1357
MGS	3072	3072	3827	3724

Note. Orig/1 and Orig/4 denote OpenMP/original with one and four processes on a node, and Thrd/1 and Thrd/4 denote OpenMP/thread with one and four threads on a node, respectively.

mapping alone reduces the number of `mprotect` operations by 1.2–1.8 times, and the comparison between Thrd/4×4 results and the Thrd/16×1 results shows that the use of multithreading reduces the number of `mprotect` operations by an additional 1.5–5 times.

Table III further compares the average cost of an `mprotect` operation in the original and the thread versions. It shows that the average cost of an `mprotect`

TABLE IV

Number of `mprotect` Operations in the OpenMP/original and OpenMP/thread Versions of the Applications with 16-way Parallelismg on Four Nodes

Application	Orig/16 × 1	Thrd/16 × 1	Thrd/4 × 4
Barnes	320743	207293	107114
3D-FFT	97690	78040	50588
CLU	29015	16555	8308
Water	102879	60667	27691
SOR	6037	4840	969
TSP	11628	8211	5438
MGS	86154	48313	21118

Note. Orig/16 × 1 denotes OpenMP/original with four processes on a node, and Thrd/16 × 1 denotes OpenMP/thread with four processes on a node, each of which has one thread, and Thrd/4 × 4 denotes OpenMP/thread with one process on each node, with four threads.

in OpenMP/thread with four threads per node is greater than in OpenMP/original with four processes per node. The difference ranges from a low of 1.8 times for Water to a high of 10.3 times for 3D-FFT. The order of magnitude increase in the cost of `mprotect` operations coupled with the small reductions in the number of messages, the amount of data, and the number of `mprotect` operations in the OpenMP/thread version of 3D-FFT explains the slight slowdown mentioned above.

The increase in `mprotect` cost is almost entirely a result of the data structure used by AIX to represent a virtual address space: an ordered, linked list recording the allocated regions of the address space. AIX optimizes accesses to entries in the list by starting the traversal from the previous entry accessed. This optimization is, however, not very effective on multiprocessors because different threads tend to access distinct regions of memory.[3]

Figure 5 compares the number of page faults that occured in the original and the thread versions of the applications, normalized with respect to the thread versions. The detailed numbers are listed in Table III. The comparison shows that multithreading reduces the number of page faults: while the number of page faults incurred by OpenMP/thread with one thread per node and OpenMP/original with one process per node is the same, the OpenMP/thread programs with four threads per node incur 1.2–5 times fewer page faults than their OpenMP/original counterparts with four processes per node. This reduction comes from two sources. First, for multiple-reader pages, only one of the threads on a node needs to fault in order to update the page and make it accessible by all of the threads, whereas each of the processes on a node has to fault once to update its own copy. Second, using multithreading eliminates the faults required by a process accessing a page that was invalidated by another process within the same node.

Finally, Table III shows that for 16-way parallelism on four nodes, OpenMP/thread creates 1.03–5 times fewer diffs than Open MP/original.

[3] Every other brand of Unix, including Linux, to which we have ported TreadMarks, uses data structures that handle `mprotect` operations more efficiently.

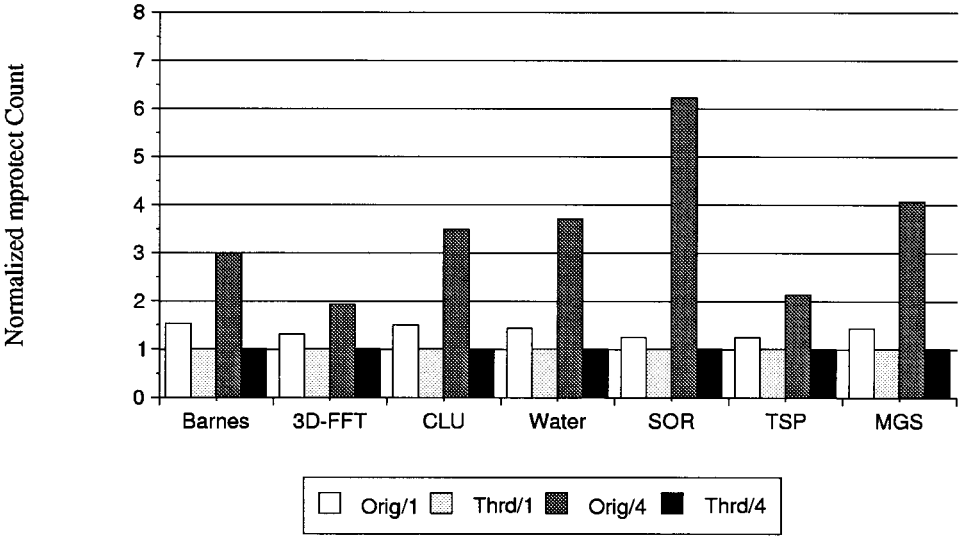

FIG. 5. Normalized page fault count comparison between the OpenMP/original and OpenMP/thread versions of the applications on four four-processor SMP nodes. Orig/1 and Orig/4 denote OpenMP/original with one and four processes on a node, and Thrd/1 and Thrd/4 denote OpenMP/thread with one and four threads on a node, respectively. Orig/1 and Orig/4 are normalized with respect to Thrd/1 and Thrd/4, respectively.

5.3.2. *OpenMP versus MPI.* A previous study comparing SDSM with message passing [10] has shown that, in general, SDSM programs send more messages and data than message passing versions due to the separation of synchronization and data transfer, the need to handle access misses caused by the use of an invalidate protocol, false sharing, and diff accumulation for migratory data. In our experiments, the OpenMP/original programs sent between 5.3 and 2568 times more messages than their MPI counterparts, for which only off-node messages are counted. The difference was the least for FFT and the most for Barnes. The reason that the MPI version of Barnes sends so many fewer messages is that it replicates the particles and duplicates rebuilding the tree by every process. As a consequence, within an iteration, the only communication by each process is a single broadcast of all the particles modified by that process. Except for SOR, the amount of data sent by OpenMP/original ranges from 1.27 times more for 3D-FFT to 93 times more for TSP, compared with off-node data in the MPI programs. For SOR, because a large percentage of the elements remains unchanged, and because TreadMarks only communicates diffs, the OpenMP program sends 15.5 times less data than the MPI code, which always communicates whole boundary rows. As has been demonstrated by Dwarkadas *et al.* [6], many of the causes of the gap in data and message count between SDSM and MPI can be overcome with additional compiler support, which is currently not present in our translator.

Our results with the OpenMP/thread programs show that on SMP nodes using multithreading in SDSM can significantly reduce the above gaps in the number of messages and the amount of data transmitted between SDSM and MPI programs. Compared to MPI, the OpenMP/thread programs send from 1.5 times more messages, for MGS, to only 271 times more messages, for Barnes. Similarly,

OpenMP/thread sends 2–28.6 times less data than MPI for two out of the seven applications, about the same amount of data for the other three and only 1.6–21.2 times more data than MPI for the remaining two.

6. RELATED WORK

Previously, we developed support for OpenMP programming on networks of single-processor workstations through a compiler that targets TreadMarks [11]. Our experiments showed that the OpenMP versions of the five selected applications achieve performance within 17% of their handwritten TreadMarks counterparts, suggesting that the compiler and the fork-join model incur very little overhead.

We are aware of five implementations of SDSM on networks of SMPs [7, 13, 14, 17, 18].

The SMP-Shasta system [14] extends the base Shasta system [15] to support fine-grain shared memory across SMP nodes. The base Shasta implements eager release consistency and a single-writer protocol with variable granularity by instrumenting the executable to insert access control operations (in-line checks) before shared-memory references. Novel techniques are developed to minimize the overhead of in-line checks. Moving Shasta to clustered SMP nodes to use hard shared memory within each node is complicated because the in-line checks are non-atomic with respect to loads and stores of shared data. The solution in SMP-Shasta uses explicit (downgrade) messages to synchronize processors within a SMP node in addition to using internal locks for all protocol operations on shared data blocks.

In their paper, Erlichson *et al.* [7] present a single-writer sequential consistency implementation and identify network bandwidth as the bottleneck. Earlier work (e.g., [5]) has demonstrated that the performance of such a system can be poor when false sharing occurs. Finally, our implementation is a relatively portable user-level implementation, while theirs is a kernel implementation specific to the Power Challenge Irix kernel.

Cashmere-2L [18] uses Unix processes instead of POSIX threads. Therefore, it must perform the same `mprotect` in each process's address space. The reason is that `mprotect` only applies to the calling process's address space, even if the underlying memory is shared between address spaces. The protocol implemented by Cashmere-2L also takes advantage of the Memory Channel network interface unique to the DEC Alpha machines.

Samanta *et al.* [13] present an implementation of a lazy, home-based, multiple-writer protocol across SMP nodes. Similar to Cashmere-2L, their implementation uses Unix processes instead of POSIX threads.

Similar to the multithreaded TreadMarks, the Brazos system [17] also uses multiple threads to take advantage of the hardware shared memory within each SMP node on a cluster of SMP nodes running Windows NT. However, the integration of the two-level consistency protocols (within and across SMP nodes) was not explicitly addressed. Speight *et al.* [16] also implemented a subset of OpenMP APIs on top of the Brazos system.

7. CONCLUSIONS

In this paper, we present the first system that implements OpenMP on a *network* of shared-memory multiprocessors. This system enables the programmer to rely on a single, standard, shared-memory API for parallelization within a multiprocessor *and* between multiprocessors. The system is implemented via a translator that converts OpenMP directives to appropriate calls to a modified version of TreadMarks that exploits the hardware shared memory within an SMP node using POSIX threads.

Using the hardware shared memory within an SMP node can significantly reduce data and messages transmitted by A. SDSM. In our experiments, the translated multithreaded TreadMarks codes send from a low of 26% less data and 29% fewer messages to a high of 9.1 times less data and 8.4 times fewer messages for our collection of applications than the translated single-threaded TreadMarks counterparts. As a consequence, they achieve up to 30% better speedups than the latter for all applications except 3D-FFT, for which the thread version is 8% slower than the process version. The slowdown in 3D-FFT is due to AIX's inefficient implementation of page protection operations. Overall, the speedups of multithreaded TreadMarks codes on four four-way SMP SP2 nodes are within 7–30% of the MPI versions.

REFERENCES

1. S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng, The SUIF compiler for scalable parallel machines, *in* "Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing," Feb. 1995.
2. C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, TreadMarks: Shared memory computing on networks of workstations, *IEEE Comput.* **29** (Feb. 1996), 18–28.
3. D. Bailey, J. Barton, T. Lasinski, and H. Simon, "The NAS parallel Benchmarks," Technical Report TR RNR-91-002, NASA Ames, Aug. 1991.
4. D. R. Butenhof, "Programming With POSIX Threads," Addison–Wesley, Reading, MA, 1997.
5. J. Carter, J. Bennett, and W. Zwaenepoel, Techniques for reducing consistency-related information in distributed shared memory systems, *ACM Trans. Comput. Systems* **13**, 3 (Aug. 1995), 205–243.
6. S. Dwarkadas, A. Cox, and W. Zwaenepoel, An integrated compile-time/run-time software distributed shared memory system, *in* "Procedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems," pp. 186–197, Oct. 1996.
7. A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy, SoftFLASH: Analyzing the performance of clustered distributed virtual shared memory, *in* "Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems," Oct. 1996.
8. 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," pp. 15–26, May 1990.
9. A. Itzkovitz and A. Schuster, Multiview and millipage—fine-grain sharing in page-based dsms, *in* "Proceedings of the Third USENIX Symposium on Operating System Design and Implementation," Feb. 1999.
10. H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel, Quantifying the performance differences between PVM and TreadMarks, *J. Parallel Distrib. Comput.* **43**, 2 (June 1997), 56–78.

11. H. Lu, Y. C. Yu, and W. Zwaenepoel, OpenMp on networks of workstations, *in* "Proceedings of Supercomputing '98," Nov. 1998.
12. Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," Version 1.0, May 1994.
13. R. Samanta, A. Bilas, L. Iftode, and J. Singh, Home-based SVM protocols for SMP clusters: design and performance, *in* "Proceedings of the Fourth International Symposium on High-Performance Computer Architecture," Feb. 1998.
14. D. Scales, K. Gharachorloo, and A. Aggarwal, Fine-grain software distributed shared memory on SMP clusters, *in* "Proceedings of the Fourth International Symposum on High-Performance Computer Architecture," pp. 125–136, Feb. 1998.
15. D. Scales, K. Gharachorloo, and C. Thekkath, Shasta: A low overhead software-only approach for supporting fine-grain shared memory, *in* "Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems," Oct. 1996.
16. W. Speight, H. Abdel-Shafi, and J. Bennett, A integrated shared-memory/message passing api for cluster-based multicomputing, *in* "Proceedings of the 2nd International Conference on Parallel and Distributed Computing and Networks (PDCN)," Dec. 1998.
17. W. Speight and J. Bennett, Brazos: A third generation DSM system, *in* "Proceedings of the 1997 USENIX Windows/NT Workshop," Aug. 1997.
18. R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott, Cashmere-2L: Software coherent shared memory on a dlustered remote write network, *in* "Proceedings of the 16th ACM Symposium on Operating Systems Principles," pp. 170–183, Oct. 1997.
19. The OpenMP Forum, "OpenMP Fortran Application Program Interface," Version 1.0. Available at http://www.openmp.org, (Oct. 1997).
20. The OpenMP Forum, "OpenMP C and C++ Application Program Interface," Version 1.0. Available at http://www.openmp.org, (Oct. 1998).
21. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, The SPLASH-2 programs: characterization and methodological considerations, *in* "Proceedings of the 22nd Annual International Symposium on Computer Architecture," pp. 24–36, June 1995.

Y. CHARLIE HU received the B.S. from the University of Science and Technology of China in 1989, the M.S. from Yale University in 1992, and the Ph.D. from Harvard University in 1997, all in computer science. He is currently a research scientist at Rice University. His research interests include parallel and distributed systems, high performance computing, large scale N-body simulations, and performance modeling and evaluation. Dr. Hu is a member of ACM, IEEE, and SIAM.

HONGHUI LU received the B.S. from Tsinghua University, China, in 1992, and the M.S. 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, including both the compiler and the runtime system.

ALAN COX received the B.S. in applied mathematics from Carnegie Mellon University, Pittsburgh, PA, in 1986 and the M.S. and Ph.D. in computer science from the University of Rochester, Rochester, NY, in 1988 and 1992, respectively. He is an Associate Professor of Computer Science at Rice University, Houston, TX. His research interests include parallel processing, computer architecture, distributed systems, concurrent programming, and performance evaluation. Dr. Cox received the NSF Young investigator (NYI) Award in 1994 and a Sloan Research Fellowship in 1998.

WILLY ZWAENEPOEL received the B.S. from the University of Gent, Belgium, in 1979, and the M.S. and Ph.D. 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, nonvolatile memory, and system support for scalable network servers.