In the Proceedings of SC'98, Orlando, FL, November 7-13, 1998

OpenMP on Networks of Workstations

Honghui Lu†, Y. Charlie Hu‡, and Willy Zwaenepoel‡

† Department of Electrical and Computer Engineering, Rice University
‡ Department of Computer Science, Rice University

Abstract

We describe an implementation of a sizable subset of OpenMP on networks of workstations (NOWs). By extending the availability of OpenMP to NOWs, we overcome one of its primary drawbacks compared to MPI, namely lack of portability to environments other than hardware shared memory machines. In order to support OpenMP execution on NOWs, our compiler targets a software distributed shared memory system (DSM) which provides multi-threaded execution and memory consistency.

This paper presents two contributions. First, we identify two aspects of the current OpenMP standard that make an implementation on NOWs hard, and suggest simple modifications to the standard that remedy the situation. These problems reflect differences in memory architecture between software and hardware shared memory and the high cost of synchronization on NOWs. Second, we present performance results of a prototype implementation of an OpenMP subset on a NOW, and compare them with hand-coded software DSM and MPI results for the same applications on the same platform. We use five applications (ASCI Sweep3d, NAS 3D-FFT, SPLASH-2 Water, QSORT, and TSP) exhibiting various styles of parallelization, including pipelined execution, data parallelism, coarse-grained parallelism, and task queues. The measurements show little difference between OpenMP and hand-coded software DSM, but both are still lagging behind MPI. Further work will concentrate on compiler optimization to reduce these differences.

1 Introduction

The OpenMP Application Program Interface (API) [7] describes a model for parallel programming on shared memory architectures. In summary, OpenMP provides a number of compiler directives that allow a user to indicate the parts of the program that are to be executed in parallel. Directives allow a step-wise migration from a sequential program to a parallel one, independent of the availability of tools for automatic parallelization. Therefore, this approach to parallelization is highly popular among users. OpenMP appears to be attracting wide-spread support among hardware and software vendors and among application developers (see http://www.openmp.org).

OpenMP currently exists only for shared memory architectures, putting it at a disadvantage compared to MPI, which runs on both shared memory and distributed memory machines. In this paper we describe an implementation of a subset of OpenMP on distributed memory machines, and in particular on a network of workstations (NOW). Such an implementation would lend increased portability to OpenMP programs and thereby further its acceptance. We use a software distributed shared memory (DSM) system to implement a shared memory abstraction on a NOW. Our compiler targets the interface provided by that software DSM.

This paper presents our experience in targeting OpenMP to a NOW. First, we describe some aspects of the proposed OpenMP standard that make compiling it for a software DSM difficult.

These difficulties relate to the cost of synchronization on a NOW and to the difference in memory architecture between hardware and software shared memories. We suggest some simple modifications to remedy the situation. These modifications correspond to good programming practice in any shared memory environment, and therefore in our opinion do not impede programmability or performance on a hardware shared memory platform. Second, we report the performance of a prototype implementation of the resulting system. We have developed a compiler for a subset of OpenMP, based on the SUIF toolkit [1], and we target the TreadMarks DSM system [2]. The system is portable to all platforms supported by TreadMarks, which includes most common Unix and Windows NT platforms. We report performance results for five applications (ASCI Sweep3D, NAS 3D-FFT, SPLASH-2 Water, TSP, and QSORT) on a switched 100Mbps Ethernet connecting 8 PentiumPro's, and we compare them to TreadMarks and MPI performance results for the same applications on the same platform.

2 OpenMP

OpenMP [7] provides a set of directives that allow the user to annotate a sequential program to indicate how it should be executed in parallel. The directives appear as special Fortran comments. The Fortran API assumes 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. OpenMP provides three kinds of directives: parallel and work sharing directives, data environment directives, and synchronization directives. We only explain the directives relevant to this paper, and refer interested readers to the OpenMP standard [7] for the full specification. In order to support both Fortran and C, we have introduced directives for C similar to those defined in the standard document for Fortran.

The two basic parallel directives are *parallel* and *parallel do*. The *parallel* and *end parallel* directives define a *parallel region*, which is a block of code that is to be executed by multiple threads in parallel. The *parallel do* directive specifies a parallel region that contains a single *do* loop.

The data environment directives control the data environment during parallel execution. They appear at the beginning of a parallel region, immediately following the parallel directives. There are four data environment directives, *shared*, *private*, *firstprivate* and *reduction*, each of which is followed by a list of variables. Variables default to *shared*, which means shared among all the threads in a parallel region. *Private* variables have one separate copy per thread. Their values are undefined when entering or exiting a parallel region. *Firstprivate* variables have the same attributes as *private* variables, but, in addition, the private copies are initialized to the value of the corresponding variables right before the parallel region. The *reduction* directive identifies reduction variables. According to the standard, reduction variables must be scalar, but we extend the standard to include arrays. Finally, the Fortran standard provides the *threadprivate* directive for named common blocks. Variables in a *threadprivate* common block 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*. When a thread encounters a *barrier*, it waits until all of the other threads in the parallel region have reached this point. 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 the threads that entered the critical section earlier. The *flush* directive guarantees that all prior modifications to the variables named in the *flush* are seen by all threads after this point. If no variables are specified, then all prior modifications to all of memory are seen by all threads after this point.

3 Proposed Modifications to the Standard

We propose two modifications to the OpenMP standard:

1. Variables in a parallel region default to private instead of shared, or, in other words, all shared variables must be explicitly declared as such.

2. We remove *flush*, and introduce *condition variables* and *semaphores*.

3.1 Private Versus Shared

We propose to make variables default to private in the software DSM implementation of OpenMP. This modification reflects the difference between the memory architectures of software and the hardware shared memory.

On a hardware shared memory machine the entire address space is shared by all threads. Variables in statically allocated memory, such as global variables in C and common blocks in Fortran, are shared among threads. Similarly, dynamically allocated memory, such as the heap in C, is shared by all threads. Each thread has a separate stack, which is invisible to other threads by lexical scope rules. However, a variable on a thread's stack can be shared with other threads by passing them a pointer to that variable.

We suspect that the decision to make shared the default in the OpenMP standard reflects the fact that, on a hardware shared memory machine, shared variables are less expensive to implement than private variables. Global shared variables require no additional support, and local shared variables can be implemented by passing a pointer to the variable from the master to the slaves. On the other hand, private variables need some extra support. If a private variable exists only in a single parallel region, it can be implemented by redeclaring that variable so that a private copy appears on the stack of each thread. If a global private variable is to persist through the program i.e. as a result of the use of the *threadprivate* directive, a copy of the global variable has to be generated for each thread.

In contrast, in software DSMs, only part of the address space is shared. Software DSMs vary in what part of the address space is shared. In some systems, the statically allocated variables are shared, in others the heap, in still others a special shared memory allocation routine needs to be called to declare an area of memory as shared. The stack is private, and inaccessible to other threads. This design is a result of the high cost of tracking shared memory accesses in software, a cost that would quickly become prohibitive if all of memory is to be shared.

In our software DSM implementation of OpenMP, variables default to private. Since different threads have partly disjoint address spaces, private variables come for free by allocating them in the disjoint portions of the address spaces. For shared variables, the compiler must infer the actual memory locations that are shared from the shared directives, and relocate these memory locations to the shared part of the address space. If a variable is declared shared in one parallel region and private in another, the compiler resorts to the hardware shared memory solution for private variables, and redeclares the variable for the region in which it is declared as private.

By defaulting to private, shared variables have to be explicitly marked as such. It can be argued that making shared the default improves sequential portability, because the majority of variables may be shared. In addition, in the fork-join model, a shared variable can be used to pass values from

```
shared volatile int available = FALSE;
shared volatile int done = FALSE;

Producer:                           Consumer:
    write data;                         while (!available);
    available = TRUE;                   available = FALSE;
    #pragma flush                       read data;
    while (!done);                      done = TRUE;
    done = FALSE;                       #pragma flush
```

Figure 1: Pipeline implemented with flush

the master to the slaves. In our experiments, we use *firstprivate* variables for this purpose. So far, our experience shows that only a small number of variables must be marked *shared* or *firstprivate*. Moreover, since access to shared variables must be synchronized, knowing exactly what is shared helps ensuring the program's correctness. In practice, the two different approaches can be unified by requiring that all shared and all private variables be declared as such.

3.2 Synchronization Directives

OpenMP provides three synchronization directives, *critical section*, *barrier*, and *flush*. These synchronization primitives can lead to awkward programming constructs for pipelined or task-queue based parallelism. In addition, *flush* is expensive to implement on software DSMs. To mitigate these problems, we introduce *condition variables* and *semaphores*.

3.2.1 Pipeline

In a pipeline, the consumer must wait until the data has been written by the producer. The next round of the producer cannot start until the consumer finishes reading the data, because the producer may overwrite it. A producer/consumer pair can be synchronized by two shared flags, *available* and *done*, as shown in Figure 1. Both flags are initialized to false. The producer writes the data, sets the *available* flag, and flushes. On the other side, the consumer busy-waits in a while loop until *available* becomes true. The consumer then resets *available* to false, reads the data, sets *done* to true, and flushes. The producer has to spin until *done* is true, then resets it to false before going to the next iteration. In summary, threads have to busy-wait, because there is no mechanism to put waiting threads to sleep, and wake them up once a particular condition becomes true.

3.2.2 Task Queue

A task queue is another common work sharing construct. Although the details may differ from one application to another, many have the general structure depicted in Figure 2. The *EnQueue* operation adds a task to the task queue, and the *DeQueue* operation removes one. The DeQueue subroutine returns a pointer to the task, with a null pointer indicating the end of the program. If the task queue is empty when a thread tries to dequeue a task, the thread waits either until the task queue becomes non-empty, or until all threads are waiting for tasks, indicating the end of the program. The program uses a shared counter *nwait* to keep track of the number of waiting threads. A thread increases the counter by one before waiting, and decreases the counter by one after having resumed the computation. A thread needs exclusive access to the task queue and the counter in order to modify them.

Figure 2 shows the implementation of the *EnQueue* and the *DeQueue* operations using critical sections and flush. The *EnQueue* is protected by a single critical section. However, the *DeQueue* operation employs two critical section directives to allow the thread to wait outside any critical section, so that other threads are able to update the queue. A thread flushes after adding a task to the queue and after incrementing the counter. Again, the solution requires busy-waiting. In this particular case, we have a critical section inside the busy-wait loop. In addition, the queue may be a complicated data structure, in which case flushing it may be expensive.

3.2.3 New Synchronization Directives

We propose to remove *flush*, and we introduce semaphores and condition variables. Both are powerful synchronization tools well known in the operation system textbooks (see, for example, [10]). The condition variables are included in the POSIX Pthreads standard [4]. Semaphores and condition variables are suitable for different applications. Returning to our examples, semaphores are suitable for pipelines, and condition variables for task queues. In both cases, the new synchronization primitives allow a simpler expression of the problem and a more efficient implementation than using *flush*.

Semaphores A semaphore S is a shared integer variable that, except for initialization, is accessed only through two standard atomic operations: *sema_wait* and *sema_signal*. The classic definitions of *sema_wait* and *sema_signal* are:

```
sema_wait(S): while (S <= 0) do no-op;
              S--;

sema_signal(S): S++;
```

It is guaranteed that a thread completing *sema_wait* on a semaphore sees the updates of all the threads that have previously issued a *sema_signal* on the same semaphore. An implementation can avoid busy-waiting by blocking the waiting thread and putting it in a queue. A *sema_signal* wakes up one waiting thread, if any.

Condition Variables Condition variables must be used within critical sections. They are used to atomically block threads until a particular condition is true. There are three primitives:

```
cond_wait(id):        Block on a condition variable
cond_signal(id):      Unblock one waiting thread
cond_broadcast(id):   Unblock all waiting threads
```

A *cond_wait* blocks the calling thread until a corresponding *cond_signal* is issued by another thread. The *cond_wait* also causes the thread to exit the critical section, so that other threads can enter and change the shared variables. A *cond_signal* unblocks one thread waiting on the same condition variable within the same critical section (any critical section with the same name). In contrast to the signal in semaphores, *cond_signal* has no effect if no thread is waiting. A *cond_broadcast* signals all the waiting threads. Upon wakeup, a thread contends for access to the critical section, and, when successful, resumes its execution from the statement after the *cond_wait*.

```
void EnQueue(TASK_TYPE *task)
{
    #pragma critical
    {
        add a task to QUEUE;
        if (nwait > 0)
            #pragma flush (QUEUE)
    }
}

TASK_TYPE *DeQueue()
{
    TASK_TYPE *task = NULL;  /* A private variable. */

    #pragma critical
    {
        if ( QUEUE != NULL )
            task =  Remove from task queue;
        else {
            ++nwait;
            if (nwait == nthreads)
                #pragma flush (nwait);
        }
    }

    while ( task == NULL && nwait < nthreads ) {
        if (QUEUE != NULL) {
            #pragma critical
            if (QUEUE != NULL) {
                task = Remove from task queue;
                --nwait;
            }
        }
    }
    return(task);
}
```

Figure 2: Task queue implemented with critical sections and flush

```
semaphore available = 0;
semaphore done = 0;

Producer:                              Consumer:
    write data;                            #pragma sema_wait(available);
    #pragma sema_signal(available);        read data;
    #pragma sema_wait(done);               #pragma sema_signal(done);
```

Figure 3: Pipeline implemented with semaphores

A pipeline can be easily implemented with semaphores, as shown in Figure 3. The flags are declared as semaphores and initialized to zero. Compared to the implementation using *flush*, busy-waiting is eliminated.

A pipeline can also be expressed with condition variables, but the code is not as concise, because the operations on the condition variables have to be within critical sections, and an additional shared variable is needed to remember the number of signals that have occurred before the wait.

A solution for the task queue problem using condition variables is shown in Figure 4. Compared with the implementation using *flush*, a *cond_signal* call replaces the *flush* after adding a task to the queue, and a *cond_broadcast* replaces the *flush* after the *nwait* counter reaches the number of threads. Only one critical section is used in *DeQueue* which protects the entire operation. Instead of the busy-waiting loop, a single call to *cond_wait* blocks the thread until a signal is issued.

One can also implement a task queue using critical sections and semaphores, but, as when using *flush*, it would require leaving the critical section to perform the *sema_wait*, and then re-entering a second critical section.

3.2.4 Performance Issues

Introducing the two new synchronization primitives not only eliminates busy-waiting, but also allows a more efficient implementation in software shared memory.

Implementing flush on hardware shared memory machines is straightforward and incurs little overhead. It suffices to write back the changes to shared variables currently in registers and issue a write barrier afterwards. It is, however, expensive to implement flush in software DSM. Without knowing which thread is waiting for the condition, the flushing thread has to notify all other threads of its modifications to the shared memory. For n threads, a total of $2(n-1)$ messages are sent, half of which are used for acknowledgments. Most of these messages are redundant, and numerous threads are interrupted unnecessarily.

Semaphores and condition variables can be implemented with a small constant number of messages, because the synchronization information only flows from the signaling thread to the waiting thread, perhaps via a third-party manager, who keeps track of the waiting threads (see Section 4.1.2).

4 Implementation

We have developed a compiler for a subset of OpenMP, based on the SUIF toolkit [1]. The compiler targets the TreadMarks software DSM system [2].

```
void EnQueue(TASK_TYPE *task)
{
    #pragma critical
    {
       add task to QUEUE;
       if (nwait > 0)
          #pragma cond_signal(0);
    }
}

TASK-TYPE *DeQueue()
{
    TASK_TYPE *task = NULL;

    #pragma critical
    {
        while (QUEUE == NULL && nwait < nthreads) {
            nwait++;
            if (nwait == nthreads)
                #pragma cond_broadcast(0);
            else {
                #pragma cond_wait(0);
                if ( nwait < nthreads )
                    nwait--;
            }
        }
        if (QUEUE != NULL)
            task = remove from task queue;
    }
    return(task);
}
```

Figure 4: Task queue implementation using critical sections and condition variables

4.1 TreadMarks Distributed Shared Memory

TreadMarks [2] is a user-level DSM system that runs on most commonly available Unix systems and on Windows NT. It provides a global shared address space on top of physically distributed memories. The parallel threads synchronize via primitives similar to those used in hardware shared memory machines: barriers, mutex locks, condition variables and semaphores. In Fortran, the shared data are placed in a common block loaded in a standard location. In C, the program has to call the *Tmk_malloc* routine to allocate shared variables in the shared heap. To support OpenMP-style environments, 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 shared memory compilers [1]. For performance reasons, all threads are created at the beginning of the execution. During sequential execution, the slave threads are blocked waiting for the next *Tmk_fork* issued by the master.

4.1.1 Memory Consistency Model

TreadMarks relies on user-level memory management techniques 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) 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 *lazy* implementation delays the propagation of consistency information until the time of an acquire. Furthermore, the releaser notifies the acquiring thread of which pages have been modified, causing the acquiring thread to *invalidate* its local copies of these pages. A thread incurs a page fault on the first access to an invalidated page, and obtains up-to-date value for that page from previous releasers.

4.1.2 Synchronization Primitives

Barrier arrivals are modeled as releases, and barrier departures are acquires. Barriers have a centralized manager. At a barrier arrival, each thread sends a release message to the manager, and waits for a departure message. The manager broadcasts a barrier departure message to all threads after all have arrived at the same 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. In the lazy release consistency protocol, the releasing threads delays the propagation of consistency data to the acquiring thread until after receiving the acquiring request.

Each condition variable is associated with a lock. The lock manager maintains a queue of waiting threads for each condition variable. On a *cond_wait*, a thread releases the lock, and contacts the manager who inserts it in the queue of threads waiting on this condition variable. A *cond_signal* also contacts the manager. If there are any threads in the condition variable's queue, the manager removes the first thread from that queue, and puts it at the end of the queue for the lock. The waiting thread will regain the lock after all previous lock acquires for the same lock are released.

A *sema_signal* corresponds to a release in the release consistency model, and a *sema_wait* corresponds to an acquire. Each semaphore has a statically assigned manager. A signaling thread sends a message to the manager including the consistency information. A thread performing a *sema_wait* also sends a message to the manager, who replies with the necessary consistency information once the waiting thread is allowed to continue. Thus a *sema_signal* or a *sema_wait* costs two messages, including an acknowledgment.

4.2 An OpenMP to TreadMarks Compiler

The compiler analysis is relatively simple, because TreadMarks provides a shared memory API on top of a workstation cluster. Since only part of the memory space is shared, the compiler has to identify the shared variables and allocate them in the shared memory. Other than this, the transformation from sequential programs to multi-threaded TreadMarks programs is straightforward.

4.2.1 Compiler Analysis for Shared Variables

The compiler analysis has two phases, where the first phase infers the actual shared locations from the directives, and the second phase finds the locations that are declared both shared and private in different parallel regions. In the absence of recursion and variable subroutine names, each can be done by one pass over the subroutines.

In the first phase, the subroutines are sorted so that a callee always appears before its callers, and the callees are examined first. If a pointer passed down the call chain is marked shared in the subroutine, this phase finds out the location it points to. An actual parameter is marked shared if the variable is passed by reference, and the corresponding formal parameter is already marked shared in the callee.

The second phase starts with the callers, and processes a caller before its callees. This phase allows the compiler to spot conflicting variable declarations in different subroutines. In this phase, if a pointer to the shared data is passed down in a subroutine call, the corresponding formal parameter is marked shared. The compiler then allocates shared variables on the shared memory. For variables marked both shared and private in different parallel regions, an error is given if the variable is a pointer. Otherwise the variable is redeclared in the parallel region in which it is marked private.

4.2.2 Compiler Transformations

Our compiler translates the sequential program annotated with a subset of OpenMP directives into a fork-join parallel program. The compiler encapsulates each parallel region into a separate subroutine. This subroutine also includes code, generated by the compiler, allowing 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 passes a pointer to this subroutine to the slaves at the time of the fork. Pointers to shared variables and initial values of *firstprivate* variables are copied

Application	Data size	Sequential Time (sec.)	OpenMP Directives	
			Parallel	Synchronization
Sweep3D	50×50×50	215	parallel region	semaphore
Water	1728 molecules	364	parallel do/region	barrier
3D-FFT	128×64×64	31	parallel do	none
TSP	19 cities	58	parallel region	critical
SQORT	1048576, bubble threshold 2048	57	parallel region	critical, condition variables

Table 1: Applications, input data sets, sequential execution time, and parallel and synchronization directives in the OpenMP versions

into a structure and passed at fork. The OpenMP synchronization directives translate directly to the TreadMarks synchronizations.

5 Applications and Their OpenMP Implementations

We use five applications in this study: ASCI Sweep3D, NAS 3D-FFT, SPLASH-2 Water, TSP, and QSORT. Table 1 summarizes the problem sizes, the sequential running times, and the parallel and synchronization directives used in the OpenMP implementations of the applications. The sequential running times are used as the basis for the speedup figures reported in the next section.

Sweep3D The Sweep3D benchmark from the DOE ASCI Blue Benchmark suite (http://www.llnl.gov/asci_benchmarks/) solves a one-group time-independent discrete-ordinates three-dimensional Cartesian geometry neutron transport problem. The main data structure is a 3-D mesh. The code uses a level of blocking along all three dimensions to achieve certain level of granularity. It then performs multiple 2-D wavefront sweeping over the 3-D blocks.

In OpenMP, the data dependence between two neighbor threads along each pipeline is expressed using our proposed *sema_signal/sema_wait* synchronization directives.

3D-FFT 3D-FFT from the NAS benchmark suite [3] 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, with both expressed as data parallel operations.

In OpenMP, the data parallelism is naturally expressed using the *parallel do* directive.

Water Water from the SPLASH-2 [11] 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. During each time step, both intra- and inter-molecular potentials are computed. The parallel algorithm statically divides the array of molecules into equally sized contiguous blocks, assigning each block to a processor. The bulk of the interprocessor communication from synchronization that takes place during the inter-molecular force computation.

In OpenMP, the evaluation of intra-molecule potentials requires no interactions between molecules and is parallelized using the *parallel do* directive. The evaluation of inter-molecule potentials can also be parallelized with *parallel do*, but to avoid excessive synchronization, we use coarse-grain parallelism, e.g., we divide the molecules among the nodes, and have one thread work on all the molecules on the same node. This level of coarse-grain parallelism is expressed using the *parallel region* 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 process repeatedly dequeues the most promising path from the priority queue, extends it by one city, and enqueues the new path, or takes the dequeued path and tries all permutations of the remaining nodes.

In OpenMP, the threads are created using the *parallel region* directive. The mutually exclusive accesses to the priority queue are expressed using *critical*. Because of the use of priority queue, the dequeue and the following enqueue operations by the same processor are actually carried out within one critical section. Therefore, there is no need to use condition variables for TSP.

SQORT Quicksort sorts an array of integers by recursively partitioning the array into subarrays, and resorting to bubblesort when the subarray is sufficiently short. Quicksort employs a task queue, wherein each task element is a pointer to a subarray. A thread repeatedly removes a subarray from the task queue, subdivides it, and puts generated tasks back to the task queue.

The OpenMP EnQueue and DeQueue operations are implemented with critical sections and a condition variable, as shown in the task queue example in Figure 4.

6 Experiments

Our experiments compare the performance of our compiler transformed OpenMP codes with that of hand-written TreadMarks as well as MPI codes.

Our experimental platform is a network of eight 166MHz Pentium Pros running FreeBSD 2.2.5 and connected by a switched, full-duplex 100Mbps Ethernet. Some basic performance characteristics of TreadMarks and MPI-CH on our platform are as follows. TreadMarks uses the UDP/IP protocol for interprocessor communication. The round-trip latency for a 1-byte message using the UDP/IP protocol is 196 microseconds on this platform. The time to acquire a lock varies from 256 to 393 microseconds. The time for an eight processor barrier is 481 microseconds. The time to obtain a diff varies from 387 to 1,225 microseconds. MPI-CH uses the TCP protocol. The empty message round trip time is 510 microseconds. The maximal bandwidth is 11.3 MB/s.

Figure 5 shows the speedup comparison on eight processors for the OpenMP, TreadMarks, and MPI versions of each application. First, the OpenMP versions of codes achieve performance within 3 - 17% of their TreadMarks counterparts, suggesting that our compiler and the fork-join multithreading model incur very little overhead. Figure 5 further shows that the OpenMP version of the applications perform within 41% of the MPI versions on eight processors. This slowdown of TreadMarks codes is explained by the fact that both OpenMP and TreadMarks send more messages and data than MPI (see Table 2). Separation of synchronization and data transfer, the use of an invalidate protocol, and false sharing contribute to this extra communication and data [5, 9]. As has been demonstrated by Dwarkadas et al. [6], many of these costs can be overcome with additional compiler support, which is currently not present in our prototype.

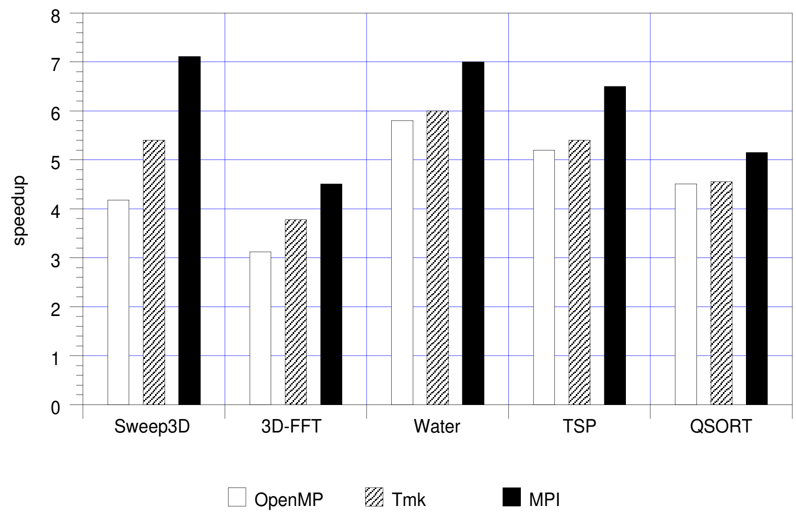

Figure 5: Speedup comparison among the OpenMP, TreadMarks, and MPI versions of the applications

Application	Data (Mbytes)			Messages		
	OpenMP	Tmk	MPI	OpenMP	Tmk	MPI
Sweep3D	128.8	129.5	46.1	85628	86453	9600
3D-FFT	80.4	73.8	51.4	51885	47495	1610
Water	81.0	82.0	9.1	59699	60842	620
TSP	29.5	29.5	0.03	23081	23190	1254
QSORT	33.0	32.5	36.9	32910	32204	3126

Table 2: Amount of data transmitted and number of messages in the OpenMP, TreadMarks, and MPI versions of the applications

7 Related Work

Cox et al. [5] evaluated the use of software DSM as the target for a parallelizing compiler on a message passing machine. They identified the factors that account for the performance differences, estimated their relative importance, and described methods to improve the performance. They used the APR shared memory parallelizing compiler (SPF), and the directives of the source programs are restricted to parallel do. Keleher and Tseng [8] also performed a similar study using the Stanford SUIF [1] parallelizing compiler to generate parallel programs for software DSM systems. Their study is also restricted to do loops.

8 Conclusions

We have demonstrated that it is possible to implement an OpenMP-like environment on a NOW. Only minor modifications to the standard are required, and these could easily be incorporated into later versions of the standard. Our prototype implementation is reasonably efficient, although still lagging behind MPI. In our further work we will focus on various compiler optimizations to reduce the performance difference between OpenMP and MPI.

Acknowledgments

This work is supported in part by the National Science Foundation under Grants CCR-9521735 and CDA-9626318.

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*, February 1995.

[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 TR RNR-91-002, NASA Ames, August 1991.

[4] David R. Butenhof. *Programming With Posix Threads.* Addison-Wesley, 1997.

[5] A.L. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel. Evaluating the performance of software distributed shared memory as a target for parallelizing compilers. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 474–482, April 1997.

[6] 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*, pages 186–197, October 1996.

[7] The OpenMP Forum. OpenMP Fortran Application Program Interface, Version 1.0. http://www.openmp.org, Octorber 1997.

[8] P. Keleher and C. Tseng. Enhancing software DSM for compiler-parallelized applications. In *Proceedings of the 11th International Parallel Processing Symposium*, April 1997.

[9] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Quantifying the performance differences between PVM and TreadMarks. *Journal of Parallel and Distributed Computing*, 43(2):56–78, June 1997.

[10] J.L. Peterson and A. Silberschatz. *Operating System Concepts.* Addison-Wesley, Reading, Massachusetts, second edition, 1985.

[11] 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*, pages 24–36, June 1995.

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, including both the compiler and the runtime system.
e-mail: hhl@cs.rice.edu *URL:* http://www.cs.rice.edu/~hhl

Y. Charlie Hu received the B.S. degree from the University of Science and Technology of China in 1989, the M.S. degree from Yale University in 1992, and the Ph.D. degree 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.
e-mail: ychu@cs.rice.edu *URL:* http://www.cs.rice.edu/~ychu

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, nonvolatile memory, and system support for scalable network servers.
e-mail: willy@cs.rice.edu *URL:* http://www.cs.rice.edu/~willy